Bachelor’s thesis

Information and Communications Technology

2021

Yasir Al-Ameri

POWER MANAGEMENT FOR A MULTI-DEVELOPER ZEPHYR ENVIRONMENT

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2021 | 51

Yasir Al-Ameri

POWER MANAGEMENT FOR A MULTI-DEVELOPER ZEPHYR ENVIRONMENT

Battery powered portable embedded devices that are limited in memory, size, and price, have been facing a significant gap between processing power and battery capacity. To diminish this gap, rigorous hardware and software power saving techniques must be implemented. The purpose of this thesis was to optimize the hardware design of a product under development by finding and repairing design flaws that caused excess power consumption. Additionally, this work aimed to implement and customize the power management framework provided by the real time operating system used in this product which is Zephyr. With this framework implementation, power management is completely carried out by the operating system, in other words, application developers who are known not to be fully cognizant of hardware design, schematics, and component data sheets, are not obliged to perform power management in their applications.

Initially, design flaws were repaired and all chips on the device under test were turned off by the software. The remaining excess current consumption was then approached by desoldering components from the board. The experimental results suggest a drastic decrease in power consumption in which the expected battery runtime increased from 30 minutes up to approximately one month of running a motion detection application.

KEYWORDS:

Zephyr, power management, RTOS, embedded systems, IoT

CONTENTS

LIST OF ABBREVIATIONS	**5**
1 INTRODUCTION	**7**
2 THEORETICAL BACKGROUND	**9**
2.1 Power management in embedded systems	9
2.2 Hardware components of embedded systems	10
2.2.1 Resistors	11
2.2.2 Integrated circuits	12
2.2.3 Processors	15
2.2.4 System on Chips	18
2.2.5 Battery technologies	18
2.3 Embedded Software	20
3 ZEPHYR RTOS	**23**
3.1 The kernel of Zephyr RTOS	23
3.2 Device drivers	24
3.3 Power management framework	25
3.3.1 System power management	25
3.3.2 Device power management	26
3.3.3 Power states and constraints	28
4 HARDWARE AND SOFTWARE POWER MANAGEMENT METHODS	**30**
4.1 Apparatus and Development Kits	30
4.2 Design flaw repairs for low power consumption	31
4.3 Power management framework implementation	34
5 RESULTS	**37**
5.1 Bare metal test	37
5.2 Device running without power management	38
5.3 Device running with power management enabled	41
6 DISCUSSION	**43**
7 CONCLUSION	**46**

LIST OF ABBREVIATIONS

API	Application Programming Interface
BGA	Ball Grid Array Package
BLE	Bluetooth Low Energy
CPU	Central Processor Unit
DMM	Digital Multimeter
DUT	Device under test
DVSF	Dynamic Voltage and Frequency Scaling
ECU	Engine Control Unit
GPIO	General Purpose Input/Output
GPOS	General Purpose Operating System (among others, Windows, Mac OS, Linux, Embedded Linux, IOS, and Android).
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
IC	Integrated Circuit
IoT	Internet of things
LDO	Low-dropout regulator
LPS	Low Power State
MCU	Microcontroller unit
MIPS	Million Instructions per Second
NFC	Near-Field Communication

OS	Operating System
PCB	Printed Circuit Board
PM	Power Management
QFN	Quad-Flatpack No-Leads is a small size and relatively low-profile plastic package used to encapsulate some integrated circuits (ICs). The size range is currently between 1x1 mm^2 to 12x12 mm^2 (NXP, 2021).
QFP	Quad Flat Package
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System
SCL	System Clock Line
SDA	Serial Data Signal
SoC	System on Chip
SOP	Small Outline Package

1 INTRODUCTION

The emergence of Inexpensive, low power, and fast Microcontroller Units (MCUs) paved the way for small multi-purpose embedded systems. Notable among these systems are the increasingly demanded battery driven portable electronics, including healthcare monitoring devices and consumer hand-held gadgets. These devices are currently packed with a variety of different features; yet they are required to have long charge or battery change intervals. Unfortunately, improvements in battery technologies are very slow (Remler, et al., 2020). Consequently, these small electronics, as in all devices powered by batteries, are required to be power efficient. Additionally, small devices are usually also memory constrained and are mostly designed to serve real time applications. Therefore, General Purpose Operating Systems (GPOSs) that provide numerous power management standers including Linux (Vaddagiri, 2004), are no longer an option for such devices. On the other hand, Real Time Operating Systems (RTOSs) that are suitable for power and memory constrained embedded systems lack an easy-to-use power management scheme; in fact, most RTOSs do not support low power states at all (Simonović & Saranovac, 2013, p. 199).

While putting a device into low power states dynamically requires a good understanding of board design, hardware schematics, and item datasheets, in general, software developers are often unfamiliar with hardware components. Moreover, Hardware Design Documents (HDD) are mostly kept confidential, in other words, programmers outside the company might not necessarily have access to such information (Dubois, 2018).

The aim of this thesis is to explore possible methods that help reduce the power consumption of a device under development that runs an open source RTOS called Zephyr. The objectives of this thesis are first, to reduce the current consumed by the device, which is forced into a deep sleep state (all peripherals shutdown) by software. Second, to implement and customize the power management framework natively provided by Zephyr.

Numerous studies have been published on power management for resource constrained embedded devices that run an RTOS; some of which focused solely on creating the whole firmware from scratch such as in, “Low Power Firmware Design in Embedded Systems” (Mishin, 2017). Whereas the focus of other publications was on developing a power management framework in a rudimentary RTOS, as in “Power Management in

ARM Cortex M0+ with Real Time Operating System" (Liljasto, 2019). Additionally, some recent theses were found to make use of advanced RTOSs, including, "Narrowband-IoT Power Saving Modes" (Gabelle, 2021), in which Zephyr RTOS was utilized on an nRF9160 development kit, a board with flawless hardware design, in order to carry out the measurements. However, the main focus of this thesis is on a device that consists of multiple different sensors and supports an array of smart features, for which reason an advanced RTOS, that inherently provides such functionalities, was found to be the best choice for this project. Furthermore, the device under development is vitiated by many design flaws, therefore, hardware optimizations are of paramount importance for this work.

This thesis is structured as follows:

Chapter 1 introduces the power management challenge for small embedded systems and outlines the aim as well as the objectives of this thesis.

Chapter 2 provides the reader with a brief theoretical background about embedded systems and battery technologies.

Chapter 3 presents the major components of Zephyr RTOS.

Chapter 4 describes the tools and methods used in this work.

Chapter 5 presents the results achieved form different power states.

Chapter 6 discusses the work carried out and highlights some limitations as well as possible improvements.

Chapter 7 summarizes this thesis.

2 THEORETICAL BACKGROUND

In order to properly understand how to reduce power consumption in compact embedded systems, it might be useful to go through the basic concepts of power management in embedded systems, as well as the most common battery technologies used to power devices that lack a direct connection to a constant power source.

2.1 Power management in embedded systems

Embedded systems are electronic devices with an embedded software, designed to perform specific tasks. However, according to Tammy Noergaard “The definition of embedded system is fluid and difficult to pin down” (Noergaard, 2005, p. 5).

Despite the numerous possible definitions of embedded systems, all electronics require a power management subsystem either to decrease the power consumption in devices that run off batteries, or to reduce the possibility of overheating that is a well-known contributor to component failure in all electronics. Nevertheless, power management requirements in different embedded systems might vary substantially. For instance, non-portable embedded devices that are constantly connected to the power grid or other available power supplies might also have a separate cooling system connected to the same power source, therefore simple power management subsystem designs might be adequate. These devices include embedded systems in home appliances and car Engine Control Units (ECUs). Likewise, non-portable embedded systems that lack a direct connection to the mains electricity such as, IoT devices deployed in hard-to-reach areas, and industrial IoT devices, are usually configured to collect data or perform certain tasks at defined intervals, for which reason a complex power management framework for such devices might not be required (Tronicszone, 2020; Malewski, et al., 2018).

In addition to non-portable embedded systems, there are portable embedded systems, which are further divided into two categories. The first category includes devices that are relatively large such as, laptops, tablets, and cell phones. These devices are large enough to carry batteries of considerable sizes and enough memory to run a GPOS such as, Windows, iOS, or Android. These operating systems are able to run sophisticated power management algorithms, in which application and OS level power management schemes are implemented (Abdelmotalib & Wu, 2012).

The second category consists of portable devices that are smaller than the devices from the first category and are usually memory constrained. As a result, a GPOS is not an option for devices in this category. Moreover, nearly all such devices have strict time requirements. Therefore, a Real Time Operating System (RTOS) might be needed. Unfortunately, unlike GPOS, most RTOSs available today must be configured in order to run properly and save power at the same time. These configurations include device drivers and task priorities, among others. As a result, software developers are required to possess hardware skills that enable them to implement low power modes in the driver for some chips that support it, understand which chip must be powered on first, and in which order chips are to be turned off in order to save power without dead locks, along with other settings. In other words, a simple dynamic low power application might be time consuming to implement for application developers (Elvstam & Nordahl, 2016).

2.2 Hardware components of embedded systems

A basic model that describes embedded systems at higher levels of abstraction is the embedded system model illustrated in Figure 1. This model suggests that the most important layer of an embedded system is the hardware layer, in which all electronic components are installed (Noergaard, 2005, p. 12).

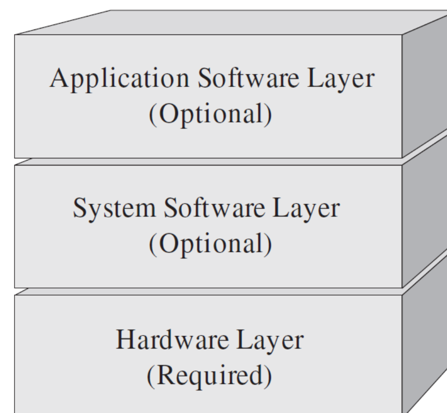


Figure 1. Embedded system model (Noergaard, 2005, p. 12).

Electronic components are divided into passive components, active components, or a combination of both (Noergaard, 2005, p. 86). These components react differently to power thus, impose energy efficiency limits in embedded systems (Tooley, 2006).

In order to understand some of the design flaw repairs carried out in this work it might be useful to go through some of these components.

2.2.1 Resistors

Resistors are passive electronic components that are made of conductive materials with different ratios of impurities. These impurities resist electric power by transforming it into heat. The dissipated heat leads to an increase in power consumption; accordingly, energy efficiency must be considered at earlier stages of the device design (Tooley, 2006).

Electronic circuits usually consist of many resistors that are used as pull-up or pull-down resistors, voltage dividers, signal conditioners, along with other uses (Braza, 2020; Sinha, 2008). Both pull-up and pull-down resistors are generally used to prevent digital inputs from obtaining undefined logic states (Braza, 2020).

The use and value of a resistor might have a great impact on power consumption in electronic circuits. For instance, the power consumed by each resistor in a simple voltage divider configuration is given by:

$$P = \left(\frac{Vin}{R1 + R2}\right)^2 . Rx$$

Equation 1. Power consumption of each resistor in a voltage divider network (Tooley, 2006).

Where “P” is the power dissipated by each resister in a voltage divider network, measured in watts. “Vin” is the maximum input voltage. “R1” and “R2” are the resistors used to divide the input voltage. “Rx” represents the resistor for which the power consumption is calculated (“R1” or “R2”).

It is clear from Equation 1 that the amount of power consumed by each resistor is directly proportional to both the input voltage and to the resistor under measurement. Thus, high values of Rx results in high power consumption.

2.2.2 Integrated circuits

Electronic circuits are commonly made up of electronic components that are ideally soldered on Printed Circuit Boards (PCBs). These PCBs usually carry immensely large numbers of passive components such as, resistors, capacitors, and inductor, as well as active components including, amplifiers and transistors. It is prominent that the size of a PCB determines the minimum possible size of the end product. As a result, electronic circuit designers of size constrained embedded devices, utilize sets of monolithic integrated circuits that are packed with an airtight semiconductor material forming what is commonly known as an integrated (IC) or a chip (Shepherd, 2002, p. 21). This hermetically sealed chip often contains transistors, resistors, and capacitors and are formed into a variety of different shapes and sizes (Shepherd, 2002, p. 203). Some common chip packages include Small Outline Package (SOP), Quad Flat Package (QFP), Quad Flat Non-leaded Package (QFN), and Ball Grid Array Package (BGA) as shown in Picture 1 (Pinsheng Electronics, 2021). It is clear from Picture 1 that the QFN and BGA packages cannot be directly probed with an oscilloscope. Applications of ICs include IO expanders, sensors, processors, audio codecs, multiplexer, and LED drivers.

Picture 1. IC packages (Pinsheng Electronics, 2021).

It may be prudent for embedded software developers to understand how ICs work and how to read data sheets as well as schematics for several reasons. Firstly, most chips implement internal pull-up and pull-down resistors that are required to be either pulled up or down by the software before the chip is powered in order for the chip to function

properly. Secondly, IO expanders for example, provide many General-Purpose Input/Output (GPIO) peripherals, all of which must be initialized to achieve low power consumption. Thirdly, some chips provide low power states that are disabled by default, to implement these states a software developer must follow a certain cycle given by the data sheet of that chip (Analog Devices, 2015, p. 43). Finally, different chips have different read and write pattern requirements, it is important to implement these procedures correctly to avoid undefined outputs. Figure 2 shows an example of the write cycle that must be followed to write temperature information to an LED driver.

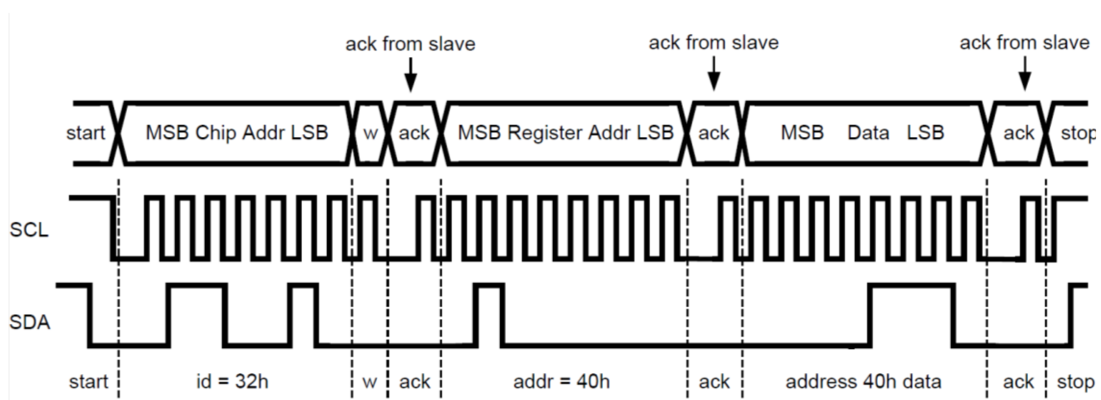


Figure 2. Write pattern required to write temperature data to register 40h for an LP55231 LED driver (Texas Instruments , 2013, p. 19).

In Figure 2 the first 8 bits represent the address of the IC. This address is used to communicate with the chip through the device driver. The address for this chip is 32 bits in hexadecimal format which is calculated by the conversion of the binary representation of the Serial Data Signal (SDA), where a logic low is zero bits and a logic high is 1 bit at each rising edge of the System Clock Line (SCL), in this case the first 8 bits of the SDA line are 00110010. The next three bits after the first 8 bits are the write or read denoted as “w” or “r” respectively, and the “ack” which is the acknowledgment received from the slave. Additionally, Figure 2 shows the address of the chip to which the data is to be written to. The read cycle on the other hand, requires a proceeding write function as shown in Figure 3 (Texas Instruments , 2013, p. 19).

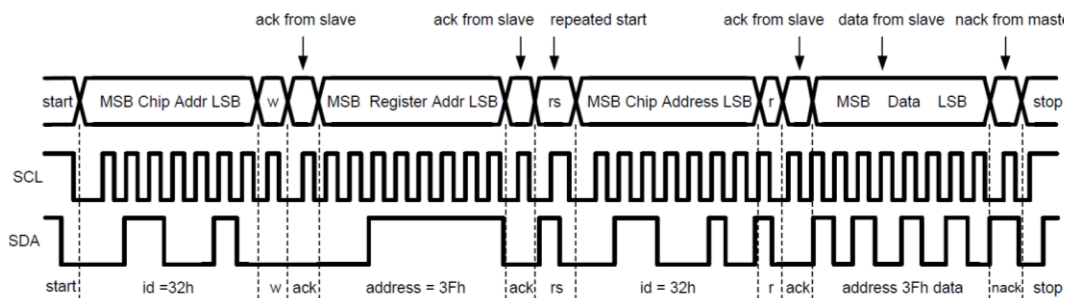


Figure 3. Read pattern that reads temperature data from register 3Fh (Texas Instruments , 2013, p. 19).

After an “ack” is received from the slave followed by a repeated start “rs” the read procedure may be started. Such patterns are not just useful to enable certain power states or manipulate some chip features but are also of great importance for software debugging as well. For example, a write attempt might fail due to a wrong address caused by a software bug that in most cases requires hours if not days to fix, whereas an oscilloscope can be used to probe the SCL and SDA lines from the chip and compare the obtained pattern and values with the data sheet as shown in Figure 4, in which the address 20h shown from the oscilloscope is the wrong address for this write attempt. The correct address according to the data sheet is 30h.

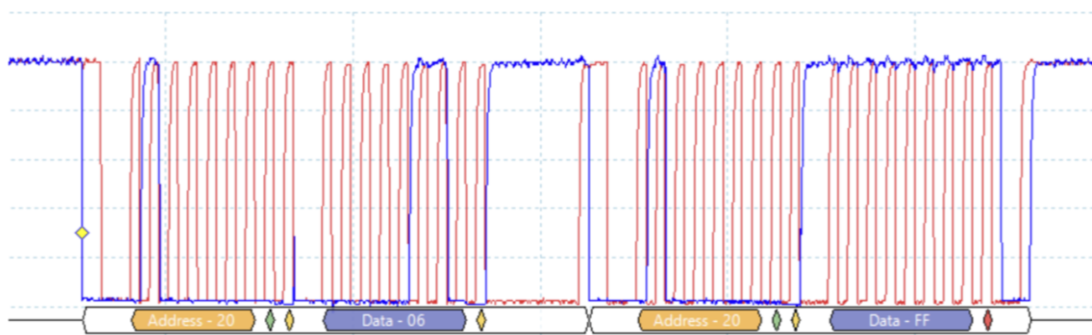


Figure 4. Oscilloscope measurement that shows a write attempt to the wrong address of an LP5036 LED driver.

2.2.3 Processors

Processors utilize transistors to perform complex tasks. Consequently, the speed and complexity of a processor depends on the number of transistors it contains. Fortunately, the transistor count in a chip followed Moore's law by doubling roughly every 18 months for at least four decades (Roser & Ritchie , 2020; Wolf, 2006). As a result, the number of transistors of an Intel CPU in the early seventies, that were only about 2300 transistors in that chip, capable of processing 0.07 Million Instructions per Second (MIPS) (Gruener & Miconi, 2018), has approached a total of 54 billion transistors in a single 7nm Graphics Processing Unit (GPU) (Walton, 2020). However, this massive growth brings two main drawbacks. Firstly, the increase of transistors in a chip lead to an increase in cost and on chip circuit complexity (Shepherd, 2002). Secondly, the power consumed by a chip increase as the number of transistors in it increases with respect to the number and complexity of the instructions executed per chip clock as shown in Figure 5 (Grochowski & Annavaram , 2006; Shalf, 2020).

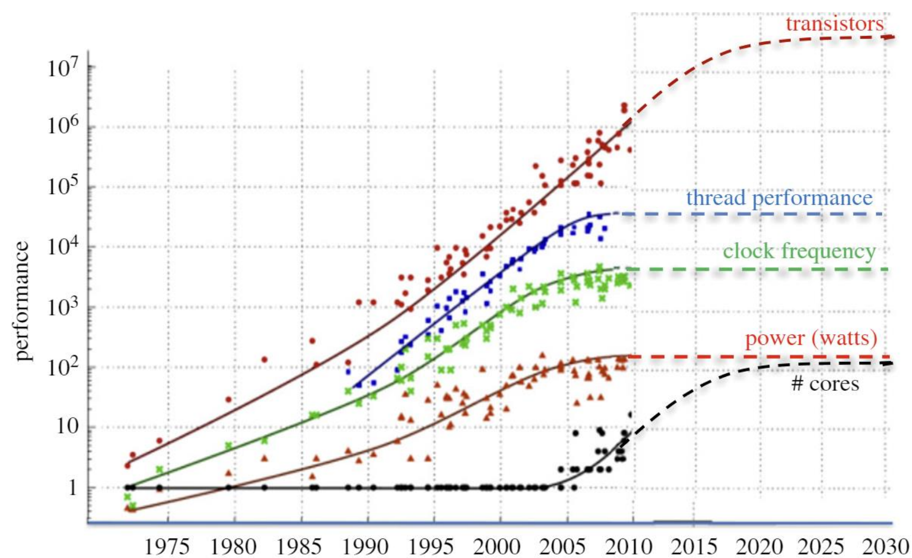


Figure 5. An increase in power consumption due to an increase in the number of active transistors and clock frequency within one clock cycle (Shalf, 2020, p.3).

It is evident from Figure 5 that transistor count, clock frequency, and power consumption are positively correlated. The consumed power is caused by three main effects, firstly, transistors, like all other electronic components, leak a certain amount of current in standby mode (Noergaard, 2005). Secondly, the instantaneous short to ground at the time the transistors switch on and off (Saxena & Akashe, 2015). Thirdly, the dynamic

power dissipation due to the charge and discharge of the transistors that are required to execute a single instruction set within one clock cycle, in other words, these transistors act as capacitors and thus the energy stored in them is given by the following formula:

$$\mathbf{W} = \frac{1}{2}CV^2$$

Equation 2. “Energy stored in a capacitor” (Tooley, 2006, p. 13).

Where: “W” is the energy stored in a capacitor in Joules (J). “C” is the capacitance. “V” is the voltage. Power is then obtained by dividing energy by time:

$$\mathbf{P} = \frac{W}{t}$$

Equation 3. Power calculated from energy (Tooley, 2006, p. 8)

Where “P” is the power in Watts and “t” is the time in seconds. However, the number of times transistors charge and discharge in a processor is given by cycles per second rather than time. Cycles per second or frequency is the inverse of time (Tooley, 2006, p. 70):

$$\mathbf{t} = \frac{1}{f}$$

Equation 4. Frequency is the inverse of time (Tooley, 2006, p. 70).

Where “f” is frequency in Hertz. The dynamic switching power is therefore the product of energy and frequency and may be formulated as:

$$\mathbf{P} = \frac{1}{2}CV^2 f$$

Equation 5. dynamic switching power in a processor (Grochowski & Annavaram , 2006).

It is clear from Equation 5 that the dynamic switching power in a CPU depends on the capacitance, which is caused by the charge and discharge of transistors per cycle, the voltage, and the number of cycles per second. Therefore, a reduction in any of these three variables should lead to a considerable drop in the overall power consumption of the processor. However, a discrete adjustment to voltage or frequency might not be an ideal solution for most applications, firstly, because some applications require higher voltages than others. Secondly, the runtime of complex instruction sets in a CPU with very slow frequencies increases, which might result in the same or even higher power consumption when compared to normal clock rates. As a result, a Dynamic Voltage and Frequency Scaling (DVFS) technique is needed in order to address different application voltage and frequency requirements. On the other hand, this DVFS power saving technique adds more complexity to the chip, which in turn increases cost. In light of these issues, chip designers in the early eighties advocated a Reduced Instruction Set Computer (RISC) architecture. The main goal of a RISC architecture was to eliminate specific seldomly used instruction sets and execute only simple instructions within one clock cycle (IBM, 2012). With this architecture it was possible to reduce the number of transistors in a processor as well as design complexity which would ultimately reduce power consumption in a CPU, as shown in Table 1 (Chen, et al., 2000).

Table 1. Remarkable improvements in CPU performance and transistor count within 40 years (WikiChip, 2021; Shirriff, 2015; Angelini & Wallossek, 2014; ARMLtd, 2019).

Processor	Architecture	Frequency	Power	Transistor count	Year
Intel 4004	CISC	740 KHz	1mW	2300	1971
Intel 8080	CISC	2 MHz	-	6000	1974
Intel 8051	CISC	-	-	50 000	1980
ARM1	RISC	33 MHz	>0.1W	25 000	1985
ARM6	RISC	33 MHz	54mW	36 000	1992

Pentium	CISC	315 MHz – 1GHz	17 W	31 000 000	1994
ARM10	RISC	300 MHz	1W	250 000	1999
Core 2 Duo	CISC	3 GHZ	65 W	230 000 000	2008
Core i7 Haswell-E	CISC	3.5 GHz – 4 GHz	18W – 124 W	2 600 000 000	2014

2.2.4 System on Chips

Small embedded systems with low power requirements typically make use of the new 8- and 32-bit Microcontroller Units (MCUs). An MCU is an IC that contains, among others, one or more Central Processor Unit (CPU), memory, and input/output peripherals. For more complex applications such as in advanced healthcare monitoring devices, an MCU combined with other chips and sensors packed into one chip which forms a System on Chip (SoC), is usually used instead of an MCU.

The encapsulation of a system in a small Quad-Flatpack No-Leads (QFN) package comes with many advantages. Firstly, some QNF packages might include two MCUs, a Bluetooth Low Energy (BLE), ZigBee and NFC, all within 7x7 mm of space (Nordic Semiconductor, 2021). Secondly, the short travel between components on the chip greatly contributes to the increase of speed and decrease in power consumption (Mujtahid, et al., 2018). On the other hand, if one component in the package fails the entire chip must be replaced. Furthermore, due to the complex designs of SoCs, extremely complicated power management frameworks are required (Lima, 2009; Malewski, et al., 2018).

2.2.5 Battery technologies

Over the last two decades great leaps in battery technologies allowed for the existence of powerful smart portable devices, such as laptops and mobile cell phones (Zhai, 2018). However, the advancement in batteries is still not fast enough to match the rapid

development of microcontroller units (Remler, et al., 2020). As a result, there is a huge gap between processor power demands and battery capacity as shown in Figure 6.

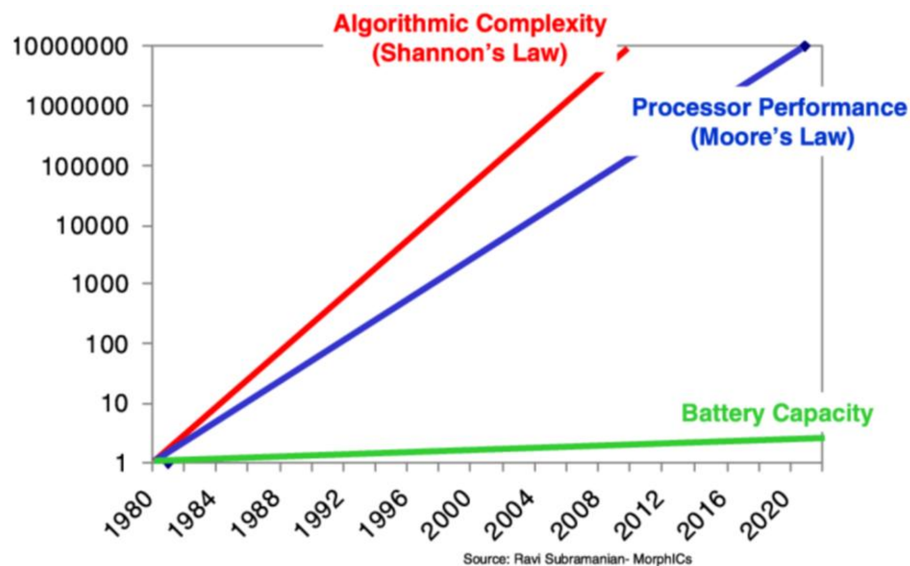


Figure 6. Improvements in battery technologies compared to CPU performance (Klasson & Hecktor, 2010).

Battery capacity is the total amount of chemical energy stored in the active material of a battery cell measured in amp-hours (Ah) (Remler, et al., 2020; Zhai, 2018). This active material is large, because it requires ions, electrolytes, cathodes, and anodes in order to transfer electric charge. So, without a significant breakthrough in material technologies the slow improvement in battery capacity might not change (Schlachter, 2013). With this faltering growth of battery capacity, knowledge of device runtime on a single battery charge becomes a necessity. For example, a device that operates on 3.7 V and draws 50 mA of current, would deplete a 3.7 V battery with a capacity of 200 mAh within:

$$t \approx \frac{C}{I}$$

Equation 6. Battery runtime estimation (Farahani, 2008).

Where “C” is the capacity of the battery measured in amp-hours, “I” is the current in amps, and “t” is the runtime in hours.

Thus:

$$\frac{200mAH}{50mA} = 4\ hours\ only$$

A battery with a capacity of 2000 mAh will increase the previously calculated runtime to 40 hours, but the price and size of the device will increase as well. Likewise, a reduction in the current consumed by the device to 5 mA will increase the runtime by 10 but might affect the overall performance of the device if not reduced properly (Malewski, et al., 2018).

2.3 Embedded Software

The Embedded System Model discussed in subchapter 2.2 suggested three main layers, a hardware layer at which all hardware components reside, and two software layers. The first software layer is the system software layer which acts as a mediator between the hardware components and the applications. This layer includes device drivers, operating systems, and middleware. The second software layer is the application layer which specifies the purpose and main tasks of the embedded device (Noergaard, 2005).

Figure 7 shows the Embedded System Model extended with three possible system software layers.

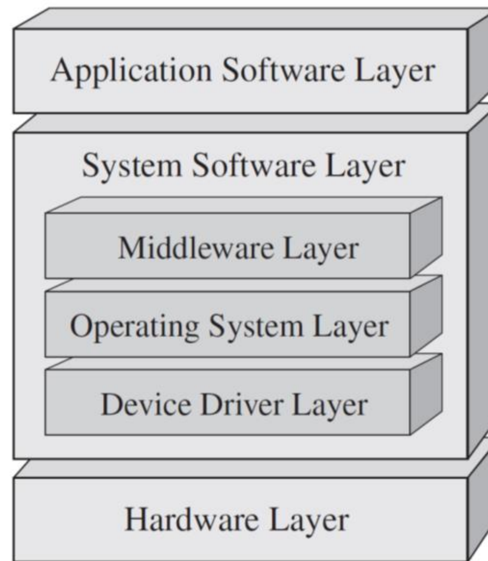


Figure 7. The Embedded System Model with possible system software layers (Noergaard, 2005, p. 383).

At its bare minimum, the system software layer must possess a device driver layer to interface the hardware components that exist on an embedded system. A device driver is a software, mostly written in C, that contains macro definitions, libraries, and functions responsible for device initialization and control (Noergaard, 2005). Since size, memory, and power constrained embedded systems that make use of SoCs require extremely complex power management frameworks, these frameworks are mostly implemented in the operating system layer (Ceolin, 2021). The OS usually used in such constrained devices is RTOS as discussed earlier in subchapter 2.1. Table 2 lists some of the most commonly used OSes for embedded devices.

Table 2. A comparison of the most popular embedded OSes (Clarysse, 2019; Pelaez, 2021; Noergaard, 2005).

OS	OS Type	License	Source model	Platform	Features
Embedded Linux	GPOS	GPL	open source	At least Cortex-A MCU (or equivalent)	Full stack

FreeRTOS	RTOS	MIT	open source	A broad range of MCUs and SoCs	Bare metal
Mbed	RTOS	Apache 2.0	open source	Cortex-M, Cortex-R, Cortex-A	Limited to ARM processors
Mynewt	RTOS	Apache 2.0	open source	Limited BSP	Full stack
VxWorks	RTOS	Proprietary	closed	A broad range of MCUs and SoCs	Full stack
Zephyr	RTOS	Apache 2.0	open source	A broad range of MCUs and SoCs	Full stack

Embedded Linux is a great option for IoT devices that can afford a powerful processor with a Memory Management Unit MMU. Likewise, Mbed and Mynewt are limited to certain platforms. FreeRTOS on the other hand supports a wide range of platforms but is bare metal, that is to say it provides no drivers, file systems, connectivity, security modules, nor a bootloader. Zephyr RTOS is therefore possibly the best choice for low power applications from the list, for two reasons, firstly, because it supports a variety of different platforms. Secondly, it includes a secure bootloader, network stack, file system, device drivers for an array of commonly used chips, and middleware (Clarysse, 2019).

3 ZEPHYR RTOS

Zephyr is an open-source Real Time Operating System (RTOS) that has gained prominence in resource constrained IoT devices. It aims to encompass all the essential software components in order to control a wide variety of hardware architectures which aids build low power applications (Clarysse, 2019; Linux Foundation, 2020). The focus of this chapter is to explore the power management framework provided by Zephyr as well as the components that interact with it.

3.1 The kernel of Zephyr RTOS

The kernel of Zephyr RTOS is an essential component of the operating system. It provides all other components of the system with services such as, Threads, a Schedular, Semaphores, Interrupts, and Mutexes. Additionally, the kernel is responsible for processes, memory, and I/O management. Basically, the kernel manages tasks according to a priority-based schedular in which threads with higher priorities (low numbers) are executed first. The thread with the lowest priority is the idle thread, which will always be scheduled when no other threads are available as shown in Figure 8 (Noergaard, 2005; Linux Foundation, 2020).

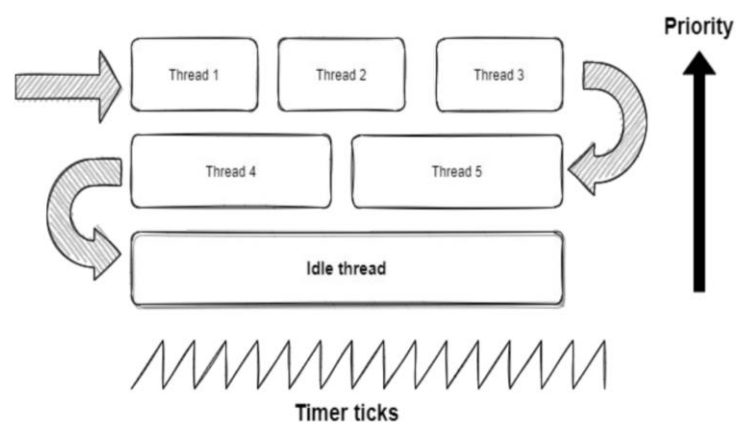


Figure 8. Zephyr kernel priority-based schedular (Ceolin, 2021, p. 10).

The thread execution flow shown in Figure 8 may be interrupted by Interrupt service Routines (ISRs) if these ISRs are not masked, or other APIs exposed by the kernel (Ceolin, 2021).

3.2 Device drivers

In Zephyr the kernel, binary linked libraries and applications are executed in one address space. The device driver model is therefore based on a shared data structure that is pre-allocated with struct devices at build time and configured with two domain-specific languages, YAML and device tree source (DTS). The YAML language is used to provide a description of the device tree overlay as shown in Picture 2, whereas the DTS language is used to define the configuration of each device instance as shown in Picture 3. This approach reduces overhead and makes the driver architecture independent (Bolivar, 2021).

```
i > 2.6.99 > zephyrproject > zephyr > samples > sensor > hmc5883l >
sample:
  name: HMC5883L Sensor Sample
tests:
  sample.sensor.hmc5883l:
    harness: sensor
    tags: sensors
    depends_on: i2c gpio
    filter: dt_compat_enabled("honeywell,hmc5883l")
```

Picture 2. A sample .yaml file for an HMC5883L sensor that describes the device tree overlay components (Bigot & Helm, 2021).

```
/*
 * Copyright (c) 2019, Linaro Ltd.
 *
 * SPDX-License-Identifier: Apache-2.0
 */

&i2c0 {
    hmc5883l@1e {
        compatible = "honeywell,hmc5883l";
        reg = <0x1E>;
        int-gpios = <&gpioc 12 GPIO_ACTIVE_LOW>;
        label = "HMC5883L";
    };
};
```

Picture 3. Device tree overlay for the HMC58831 sensor (Bigot & Helm, 2021).

The description of the device tree overlay shown in Picture 2 must match the DTS file illustrated in Picture 3. For example, the “dt_compt_enabled” and the “depends on” fields

indicate the name of the device and that it should be defined under I2C. “reg” is the hardware address of the device in hexadecimal, and the “int-gpios” is the GPIO physical pin that represents the power source to the device, which is usually pre-set either to active high, active low, disconnected, or inactive (Intel Corporation & Wind River Systems, 2021).

The third tool that is used to configure the device driver at build time is Kconfig, which is different from the DTS overlay file in that the former is used to describe the software features to be added to the final binary file, whereas the latter is used to specify the hardware components needed for device initialization, such as, the address, GPIO pins, and the required peripheral (Intel Corporation & Wind River Systems, 2021). The functions used to initialize the device, define the priority of the device, bind it to its DTS node, and implement the features it supports, are placed in a “.C” file (Linux Foundation, 2020).

3.3 Power management framework

The basic idea behind the power management framework in Zephyr is to reduce power consumption when possible. This means that the framework must turn off power to peripherals and devices that are not in use. In order to achieve this, Zephyr provides two main infrastructures which can be used in conjunction or separately (Ceolin, 2021).

3.3.1 System power management

The first infrastructure is system power management that makes use of power saving features in SoCs. In this method the system is in an active state until the schedular schedules the idle thread, then checks for power state constraints set by any component in the system, if no constraints where set, the kernel will call the policy manager which is an algorithm that places the system into a state that will save the most power based on certain conditions. The selected state will remain the current sate unless an external interrupt happens or a program times out, because the policy manager locks out all other interrupts in order to prevent preemptive threads from running (Ceolin, 2021).

The system power management method exposes several APIs to the application layer which can be used by the application to change the system state or to inhibit the system from entering certain states (Ceolin, 2021; Linux Foundation, 2020).

Figure 9 highlights the main steps that take place inside the idle thread when system power management is enabled.

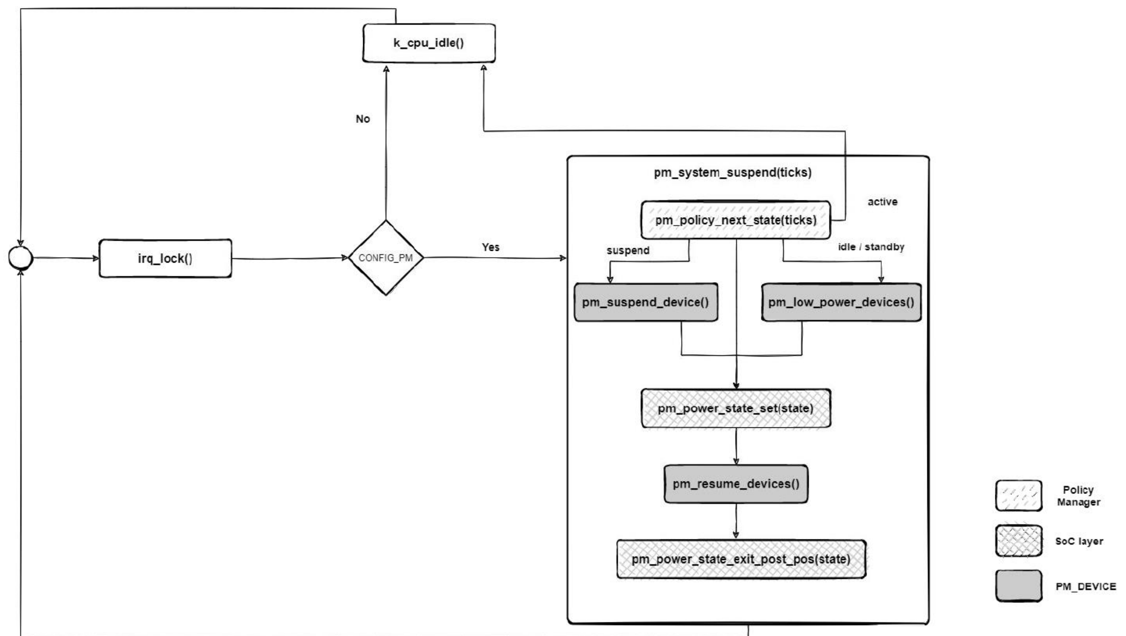


Figure 9. System power management scheme inside the idle thread (Ceolin, 2021).

When the system starts the idle thread, all interrupt requests are locked, then the system checks whether the system power management is enabled in the prj.conf file (CONFIG_PM=y), otherwise k_cpu_idle is scheduled and the power management subsystem shown in Figure 9 will not run (Ceolin, 2021).

3.3.2 Device power management

The second infrastructure that Zephyr provides to reduce power is device power management. This infrastructure consists of two possible methods, the first method is Central device power management, in which the system is prevented from entering a deep sleep state before the states of all devices are checked, if all devices are in an idle state and have been idling for a certain amount of time, then this method will check for the next scheduled event and compare it to the latency time defined in the DTS file.

Provided that all of these conditions are satisfied the system will then be allowed to enter deep sleep. The second method is Runtime device power management which relies solely on components that are power aware. These components must have the ability to put the devices used by them into a suitable power state safely. As a result, the system will enter a deep sleep state rapidly as soon as this decision is made, because it does not need to undergo all the steps required in the Central method (Linux Foundation, 2020).

For example, a system with three devices A, B, and C, where both devices A and C depend on device B as shown in Figure 10. When a program in the application layer needs to interact with device A, the device driver of device A will activate device B. Likewise, device C will require the activation of device B, at this point device B is used by two devices, in order to power it off safely device B must keep track of the devices using it, which is accomplished by the use of counters and two APIs, pm_device_get and pm_device_put. Where the “get” API will report that device B is in use and will increase the usage counter by one, whereas the “put” API will decrease the counter by one. Consequently, whenever the system makes the decision to enter a deep sleep state it only needs to check the usage count of all devices rather than undergoing all the steps executed by the Central method (Ceolin, 2021).

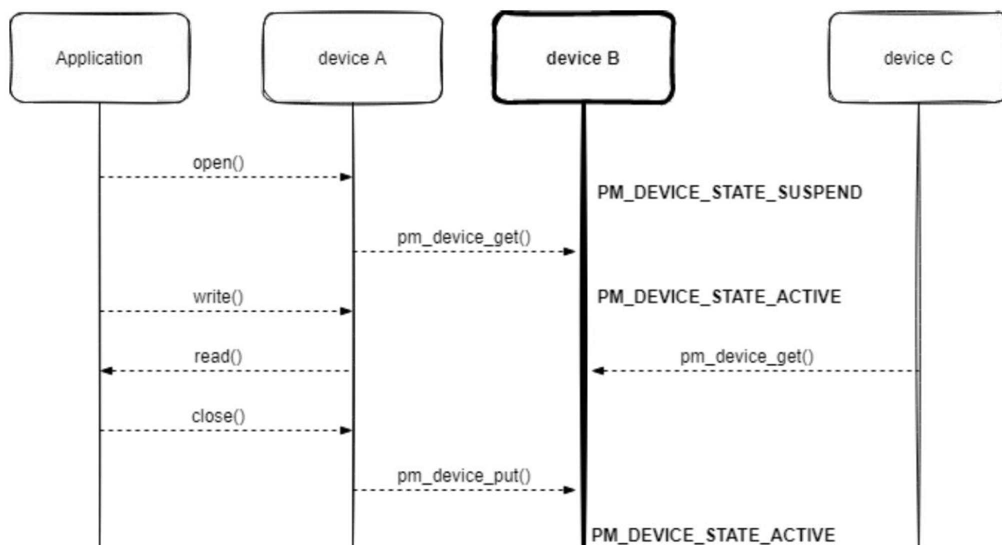


Figure 10. An example of device Runtime power management with three devices (Ceolin, 2021).

In Figure 10 device B receives two “get” calls, one from Device A and the other from device C. However, only device A makes a “put” call, therefore the usage count of device B is one and a deep sleep state is not possible.

3.3.3 Power states and constraints

The power management framework in Zephyr supports seven power states, which are given as enumerated data types, where lower numbers correspond to higher power consumption, as follows:

1. PM_STATE_ACTIVE: All peripherals are on for which reason this state consumes the most power (Linux Foundation, 2020).
2. PM_STATE_RUNTIME_IDLE: all devices retain their power states while the system enters a suitable idle state (Linux Foundation, 2020).
3. PM_STATE_SUSPEND_TO_IDLE: this state is the same as the previous state but saves more power because some peripherals are put into low power state, based on complex algorithms (Linux Foundation, 2020; Ceolin, 2021).
4. PM_STATE_STANDBY: CPUs that are not in use are turned off and peripherals are also allowed to be in low power state. This state is different from suspend to idle only in systems with multiprocessors (Linux Foundation, 2020).
5. PM_STATE_SUSPEND_TO_RAM: the power saved in this state is remarkable, because most of the system is completely powered down while the content is saved to memory which makes the wakeup latency low (Linux Foundation, 2020).
6. PM_STATE_SUSPEND_TO_DISK: the system saves all the content to a non-volatile storage and turns of the memory in addition to the power saving techniques conducted in the previous state, therefore the system will also resume with the least latency (Linux Foundation, 2020).
7. PM_STATE_SOFT_OFF: this state saved the most amount of power from all other states, however, it has the greatest wakeup latency, because all content is lost, so the system must reinitialize all devices as well as other boot requirements.

The system might be prevented from entering any of the above listed power states by constraints set at the Application layer or through any component in the System software

layer. The power management framework in Zephyr provides three APIs to set, get, or release constraints as follows (Linux Foundation, 2020):

- bool pm_constraint_get (enum pm_state state): This function checks the given power state and returns false if it is disable (Linux Foundation, 2020).

- void pm_constraint_set (enum pm_state state): This function prevents the power management framework from selecting the specified power state as the next state (Linux Foundation, 2020).

- void pm_constraint_release (enum pm_state state): When this function is called the given power state will be selected by the framework. This function must be called whenever the system is ready to enter a low power state, if a constraint has been set for this power state (Linux Foundation, 2020).

4 HARDWARE AND SOFTWARE POWER MANAGEMENT METHODS

The power management techniques were carried out on a product in its early phases of development. This product consists of two boards (PCBs) connected together via a flex cable. This product provides a variety of different functionalities that require the use of several different sensors. Some off these sensors are integrated into one chip and require a fairly complicated layout in order for it to function properly. It is not uncommon for such complex designs to have many design flaws, especially at early stages of development. These design mistakes usually require ample amount of time to find and special tools for debugging and repairing. This chapter will explore the tools used to conduct this work and most of the techniques used to reduce the power consumption in this product.

4.1 Apparatus and Development Kits

In order to implement the power management techniques on the given product, several measurement tools were needed, such as, an oscilloscope, at least two digital multimeters (DMMs), and a Power Profile Kit (PPK). Additionally, an SMD and BGA rework station as well as other tools and equipment were required for testing and design flaw repairs as shown in. Finally, a debugger was needed for cross compiling, debugging, and programing the device. The chosen debugger for this product was the nrf5340 development kit (DK). This debugger has an onboard SEGGER J-LINK which enables full programming and debugging of external targets via the Joint test Action Group (JTAG) protocol. In addition to these features the nrf5340 may be extended with the PPK for precise current measurements (Nordic Semiconductor, 2021).

Picture 4. Workshop tools and equipment, with the DUT connected to a DMM in series with a power supply.

4.2 Design flaw repairs for low power consumption

The device under test (DUT) consists of an nrf5340 SoC along with hundreds of passive and active components. Ideally, this device should consume no more than 0.09 mA with the SoC in an idle state where regulators are in a Low-dropout regulator (LDO) state, and all peripherals powered down. This is because the SoC is advertised to consume 3.3μA in this state (Nordic Semiconductor, 2021). By adding possible leakage current from all components on the DUT along with other possible hidden design flaws, 0.09 mA was found to be a rough estimate. However, the initial measurements should be taken with the SoC completely erased, to rule out any possible software bugs, this means that peripherals are not set to power off devices. While some ICs on the DUT are connected directly to the power source, these chips are usually active by default. Therefore, the current consumed by these chips should be added to the previously estimated 0.09 mA. On average such chips consume about 3.5 mA in their active state. With five chips active, the expected current consumption would then be 17.59 mA. The initial measurement obtained from the DUT was approximately 300 mA. This is almost 17 times more than the expected value. For such high current consumption, a thermal camera was able to easily spot the errors which were found to be a manufacturing mistake in which two chips were placed to the wrong direction. Similarly, several other design flaws were found, some of which had a pull-down resistor that should have been a pull-up, while others were board layout issues.

After solving the flaws that were detectable by a thermal camera, the current consumption dropped from 300 mA down to 24 mA. With all peripherals on, the expected excess current is hence around:

24 mA – 17.59 mA = 6.41 mA

At this point it is prudent to turn all peripherals off, as an attempt to power down all active devices. This was achieved by creating an application in Zephyr that will set all GPIO pins to a suitable state. The priority of the application source code file should be lower than all active device drivers in the system, this is because it is not possible to turn a device off without initializing it first. This application is defined as shown in Picture 5 below.

```
DEVICE_DEFINE(io_dev, "i/o driver", io_init, NULL, NULL, NULL, POST_KERNEL, 71, NULL);
```

Picture 5. Defining a device.

The DEVICE_DEFINE macro requires nine parameters, the first two, represent the name of the device, the third parameter, is the address to the initialization function that should run when this device is scheduled by the kernels schedular. The fourth parameter is a pointer to the power control function if power management is implemented and is NULL otherwise. The fifth and sixth parameters are possible data and configuration settings respectively, these were not needed for this simple application and therefore set to NULL as well. The seventh parameter is the initialization level, which is set to post kernel. The eighth parameter is the priority that should be set to run after all other threads. The last parameter is a pointer to a possible API, this parameter is also set to NULL, because this application is only intended for testing (Linux Foundation, 2020). An excerpt of the initialization function is shown in Picture 6 below.

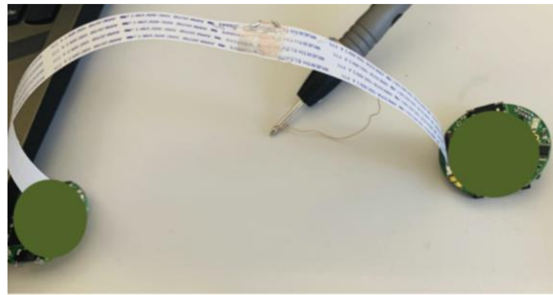
```
gpio_pin_configure(gpio_0,17,GPIO_INPUT | GPIO_PULL_UP);// AD7147_INT#
gpio_pin_configure(gpio_0,18,GPIO_OUTPUT_ACTIVE);       // AD7147_GPIO
gpio_pin_configure(gpio_0,19,GPIO_DISCONNECTED);        // NC
gpio_pin_configure(gpio_0,20,GPIO_OUTPUT_INACTIVE);     // VIBRATION_TOGGLE
gpio_pin_configure(gpio_0,21,GPIO_INPUT);               // INA220_ALERT#
gpio_pin_configure(gpio_0,22,GPIO_INPUT);               // LIS2DH_INT1
gpio_pin_configure(gpio_0,23,GPIO_INPUT);               // LIS2DH_INT2
gpio_pin_configure(gpio_0,24,GPIO_DISCONNECTED);        // NC
gpio_pin_configure(gpio_0,25,GPIO_OUTPUT_INACTIVE);     // SensorPowerEN
gpio_pin_configure(gpio_0,26,GPIO_OUTPUT_ACTIVE);       // SPEAKER_VDD_EN
gpio_pin_configure(gpio_0,27,GPIO_DISCONNECTED);        // NC
gpio_pin_configure(gpio_0,28,GPIO_INPUT);               // I2C_INT#
//gpio_pin_configure(gpio_0,29,GPIO_OUTPUT_ACTIVE);     // AD7147_BOT_CS#
gpio_pin_configure(gpio_0,30,GPIO_INPUT);               // BOT_I2C_INT#
gpio_pin_configure(gpio_0,31,GPIO_INPUT);               // BOT_GPIO_INT#

/*GPIO_1*/
```

Picture 6. GPIO pin configuration for low power consumption.

By configuring the GPIO pins to a suitable logic level, it was possible to turn off chips, that were active, from their own driver or from this same application by exposing APIs from the driver. With this configuration the current consumption was reduced drastically to 3.4 mA. With this obtained value, the predictable excessive current consumption is:

3.4 mA – 0.09 mA = 3.31 mA

Any component on the DUT may be the cause of the extra 3.31 mA current consumption. One way to tackle this endless list of possible components, is to desolder components from the DUT that have a direct path to the 3.3v power supply. This approach is mostly moot and requires access to schematics, board layouts, as well as the proper skills to carry out the process effectively. Fortunately, the author had full access to such documents together with the required skills. Since the product consists of two PCBs connected by a replaceable flex cable, it was possible to isolate the board that was causing the problem, by cutting the power line on the flex cable and placing a DMM in series with it as illustrated in Picture 7, which indeed suggested that one of the boards alone was consuming all the extra current.



Picture 7. The DUT with a flex cable in which the board-to-board power line was exposed and cut to allow probing it with a DMM.

Before the removal of any component that has a device driver, it is important to disable the driver in the prj.conf file in order to prevent the application from attempting to initialize a device that does not exist, which would cause the application to crash and thus fail to configure other GPIO pins in the system. In addition to components with device drivers, some ICs offer the option to change the slave address to allow the use of more than one IC on the same communication bus. This can be achieved by connecting the slave address pin on the PCB to the power source or to ground, usually through a pull-up/down resistor (ST-Microelectronics, 2021). Disconnecting this resistor will cause the slave address to change. As a result, the driver will not be able to bind the chip from the DTS file. Consequently, the initialization of the chip fails, forcing the application to crash, which causes an increase in current consumption.

The process of desoldering components from the PCB and applying suitable repairs for both software and hardware, resulted in a decrease in the current consumed from 3.4 mA to 0.178 mA. This is still approximately double that of the expected value. Nevertheless, with this value the DUT should idle for at least:

$$\frac{300mAH}{0.178mA} = 1685{,}4\ hours\ roughly\ 70\ days$$

4.3 Power management framework implementation

At the time of writing this document, the latest Zephyr release is 2.6.99. Unfortunately, the power management framework in this release is still under development (Ceolin, 2021). Moreover, the SoC used in the DUT does not currently support certain power states provided by this framework and were therefore bypassed in this implementation.

System power management implementation

The power management subsystem in Zephyr is disabled by default, in order to enable it, the supported power states must be declared first in the device tree as show in Picture 8.

```
power-states {
    state0: state0 {
        compatible = "zephyr,power-state";
        power-state-name = "runtime-idle";
        substate-id = <1>;
        min-residency-us = <500>;
        exit-latency-us  = <500>;
    };
    state1: state1 {
        compatible = "zephyr,power-state";
        power-state-name = "suspend-to-idle";
        substate-id = <2>;
        min-residency-us = <10000>;
        exit-latency-us  = <500>;
    };
    state2: state2 {
        compatible = "zephyr,power-state";
        power-state-name = "standby";
        substate-id = <3>;
        min-residency-us = <50000>;
        exit-latency-us  = <500>;
    };
};
```

Picture 8. Power state declarations in the device tree.

Each power state must possess a name and a minimum residency time. This time is given in microseconds and represents the minimum amount of residency time in a state

added to the time it takes to exit the state before the next scheduled event takes place, which is implemented in the residency policy as shown in Picture 9. The system power management can then be enabled by setting CONFIG_PM=y in the prj.conf.

```
if ((ticks == K_TICKS_FOREVER) ||
    (ticks >= (min_residency + exit_latency))) {
    LOG_DBG("Selected power state %d "
        "(ticks: %d, min_residency: %u)",
        pm_min_residency[i].state, ticks,
        pm_min_residency[i].min_residency_us);
    return pm_min_residency[i];
}
```

Picture 9. The minimum residency time required to enter a state.

Device power management implementation

The prototype comprises of many sensors that rely on other devices in the system such as, drivers, multiplexers, and I/O expanders. To save the most amount of power all devices, power source lines and peripherals that are not in use should be turned off dynamically during runtime. However, a device cannot simply turn off the power supply it depends on and then turn itself off. Furthermore, the power supply of one device might be in use by another device, powering it down might cause a possible transaction on the other device to fail. Therefore, such devices must be controlled by a separate component that keeps track of all devices and is responsible for turning the power source of a device off only when no other device in the system is using it. Basically, this component is a program that was added to the power management subsystem as a runtime policy manager in this implementation. It holds a list of dependencies associated with each power line used by devices. Device drivers define the power source they rely on, so that “pm_device_get” is used to notify the runtime policy manager that this power line is in use, whereas “pm_device_put” will release the power supply from this particular driver.

Device drivers may leverage runtime power management only if they implement a power control function that is pointed to in the DEVICE_DEFINE’s fourth aforementioned

parameter. The core functionality of the power control function is to return a suitable device state according to the request made by the application that is calling this function. For example, if an application needs to use a specific sensor on the prototype, without power management this sensor should have been fully initialized during boot time and never turned off. However, if power management is implemented, then this sensor might have been powered down by the runtime policy manager. As a result, the power control function must first check the current state of the sensor. If the state is off, “pm_device_get” is called to increment the usage counter of the serial communication bus that this sensor depends on. After this the sensor can be reinitialized and used by the application as shown in Picture 10.

A disadvantage of dynamic power control is the time it takes to reinitialize a device. This latency was considered both in the device tree and the runtime policy manager.

```
/* Switching from OFF to any */
if (dev->pm->state == PM_DEVICE_STATE_OFF) {
    const struct device *serial_bus = device_get_binding(DT_LABEL(DT_PARENT(DT_INST(0,sensor_name))));
    pm_device_get(serial_bus);
    /* Remove sleep mode */
    const struct device *nrst_pin = device_get_binding(cfg->rst);
    gpio_pin_set(rst, cfg->rst, 1);
    /* Re-initialize the chip */
    ret = driver_init(dev);
}
```

Picture 10. Switching a sensor on in the power control function, where rst is active low.

In addition to dynamic power management, micro delays were added to some functionalities of certain device drives, in order to imperceptibly reduce device runtime. For instance, the DUT incorporates a number of RGB LEDs which are programed to flash on and off in a certain pattern. With a slight increase in the off state as well as decrease in the time the RGBs are on, resulted in great power savings.

5 RESULTS

The ultimate goals of this research were firstly, to reduce the power consumed by the DUT from 300 mA to 0.09 mA at most. Secondly, to reduce run time power consumption by the addition of custom power states to the power management framework provided by Zephyr. This chapter presents the results obtained from different experimental stages.

5.1 Bare metal test

The bare metal test here refers to a stage at which the power management framework was not implemented. Basically, this test was carried out before and after hardware repairs took place, the measurement tool used to obtain these results was a DMM as shown in Table 3. It is worth noting that the author was responsible for conducting all of the repairs.

Table 3. Results obtained by repairing design flaws.

Current consumption (mA)	Hardware cause	Repair	Current consumption after the repair (mA)
300	Two chips soldered to the wrong direction. (Manufacture error)	Chips resoldered to the correct direction.	57
57	Layout mismatch. (Design flaw)	The pins on the board were isolated with Kapton tape, and copper wires were used to reconnect the chip.	24
24	All peripherals on.	An application created in Zephyr to configure GPIO pins for low power consumption	3.4

3.4	Wrong pull up/down resistors	The resistors were removed or replaced with the correct value and logic level.	0.178

5.2 Device running without power management

Device running without power management, in which power management was disabled from the prj.conf file. The DUT was measured by the PPK. The first measurement shown in Figure 11 was obtained from the DUT with the power completely switch off.

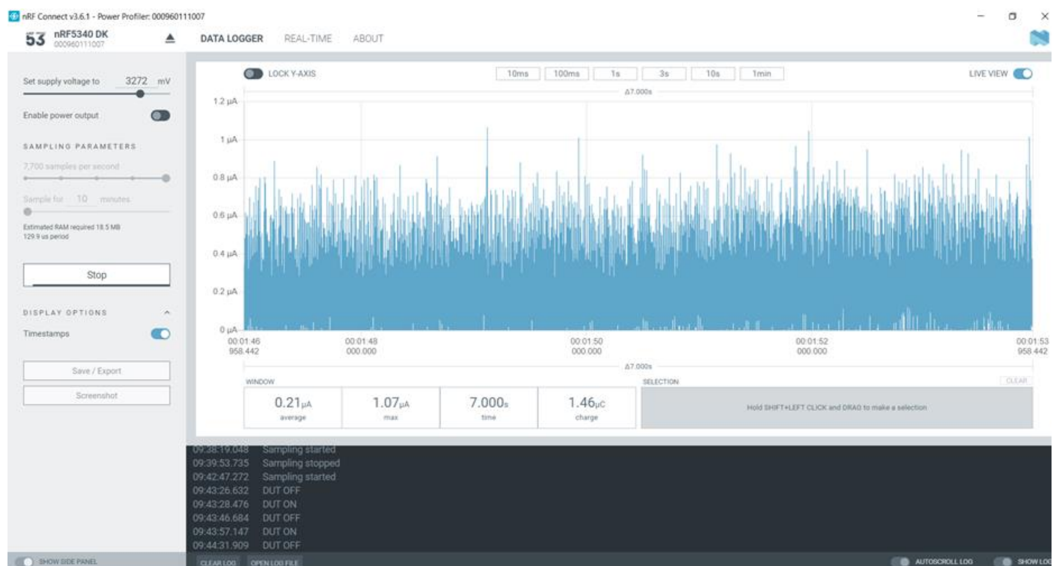


Figure 11. A screen capture from the PPK that illustrates a leakage current from the DUT even with the power switched off.

The average current that will leak from the device with the main power switched off, is shown to be 0.21 µA in Figure 11. This is a drastic reduction when compared to the initial current consumption before the design flaws were repaired.

The second test took place with an application loaded on the DUT that requires the use of the BNO08x sensor. The current consumption is shown in Figure 12 below.

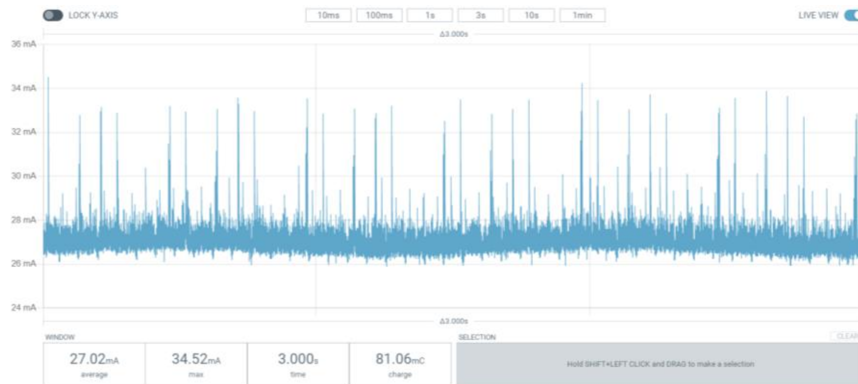


Figure 12. A screen capture from the PKK that shows the current consumed by the DUT with an application that needs to use the BNO08x sensor on the DUT in motion detection mode (this mode is rated to consume 150µA).

The average current consumption was found to be 27.07 mA. This means that a 300 mAH battery can run for only:

$$\frac{300mAH}{27.07mA} = 11{,}08\ hours$$

The third measurement was carried out on the DUT with a capacitive touch sensor on, and presumably all other devices on the DUT turned off as shown in Figure 13.

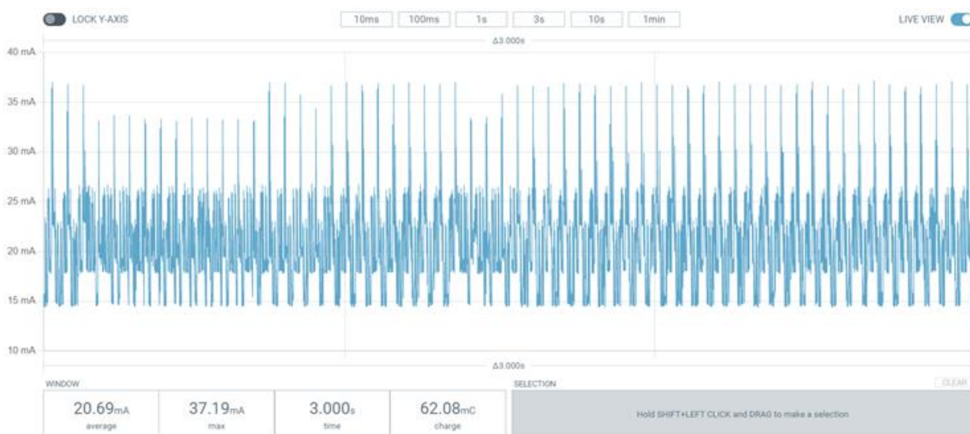


Figure 13. A screen capture from the PPK that illustrates the current consumed by the DUT with a capacitive touch sensor active and no power management implementation.

It can be seen from Figure 13 that the device is continuously turning on and off, in which the maximum current consumption is 37.19 mA. The expected maximum spike should not exceed 7 mA for this sensor.

The fourth measurement was conducted with all RGB LEDs flashing while other devices assumed to be off, as illustrated in Figure 14.

Figure 14. A screen capture of the PPK, that shows the current consumption of the DUT with all chips active and the RGB LEDs flashing without any power management enabled.

According to Figure 14, the DUT will be able to flash the LEDs for:

$$\frac{300mAH}{57.13mA} = 5{,}25\ hours$$

5.3 Device running with power management enabled

The results presented in this subsection were obtained from the DUT after the power management subsystem provided by Zephyr was enabled and customized for three devices in the system.

The first experiment was performed on the DUT with a BNO08x sensor in motion detection mode. The results are shown in Figure 15.

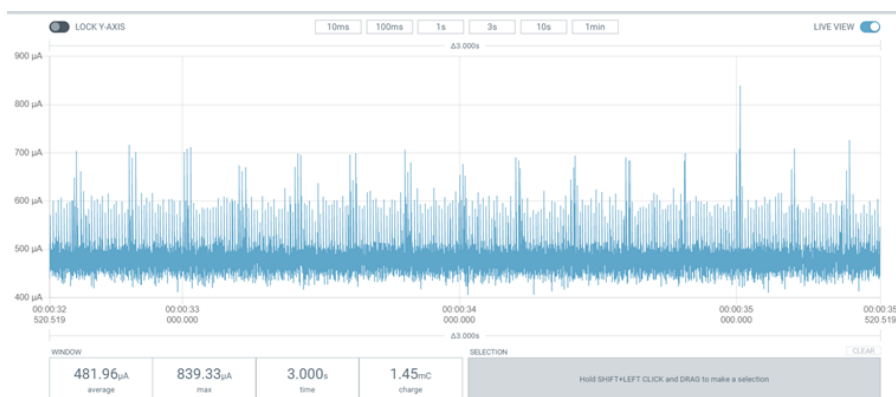


Figure 15. A screen capture from the PKK that shows a significant decrease in the current consumption of the DUT with the power management framework enabled and customized to manage a BNO08x sensor in motion detection mode (this mode is rated to consume 150μA).

Figure 15 shows an average current consumption of 481.96 mA, where the device is repeatedly turning on and off. The second experiment in this subsection was carried out on the DUT in which the power management framework was enabled and customized to control the capacitive touch sensor with all its dependencies. The obtained results are shown in Figure 16.

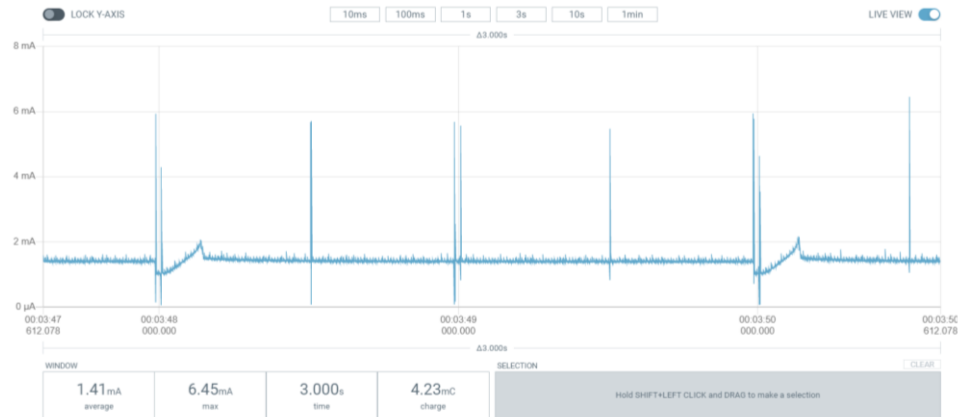


Figure 16. A screen capture from the PPK that illustrates the current consumed by the DUT with a capacitive touch sensor in a full power mode, with a custom power management implementation (the sensor is rated to consume 0.9 mA in this mode).

The final experiment in this subsection took place on the RGB LEDs. The results are shown in Figure 17. In this experiment the intensity of the RGBs was slightly reduced by configuring the device driver. Additionally, micro delays were added

Figure 17. Figure 14. A screen capture of the PPK, that shows the current consumption of the DUT with all chips active and the RGB LEDs flashing with power management enabled.

.

6 DISCUSSION

The findings from this work prove that the hardware and software methods presented in this document effectively reduced the current consumed by the device under test from a short circuit drawing 300 mA down to 0.178 mA with the device in a deep sleep state, and from an average of 27 mA, with the device running an application that requires the use of a motion detection sensor on the device, down to an average of 0.481 mA. Indeed, these results will help increase the charging intervals from less than a half an hour up to more than 26 days. Although this is a significant increase in battery life, the expected runtime for this device was much longer.

Results show that the power management framework saves the most amount of power when enabled on the capacitive touch sensor but fails to achieve the same efficiency with the BNO0x sensor. By debugging the code, it was noticed that the system fails to write to the I2C serial communication peripheral when the SoC is placed into a low power state yet succeeds to write to the same peripheral when the system is in an active state as shown in Picture 11. Additionally, the system successfully writes to the SPI peripheral in both states. Since the BNO0x sensor requires the I2C bus in this design and the capacitive touch sensor makes use of the SPI bus, it was clearly the cause of the access current consumption in the former sensor and not the case for the latter.

```
Local
> data: 0x20000244 <mux_i2c_3_data>
> cfg: 0x20000238 <mux_i2c_3_cfg>
  mux_channel: 0 '\000'
  address: 112
> data: 0x20000244 <mux_i2c_3_data>
  new_pm_state: 2
> dev: 0x1a8e0 <__device_dts_ord_125>
  ctrl_command: 1
> state: 0x20014c50 <z_idle_stacks+59128>
  cb: 0x0 <_vector_table>
  arg: 0x0 <_vector_table>
  ret: -5
Global
Static
```

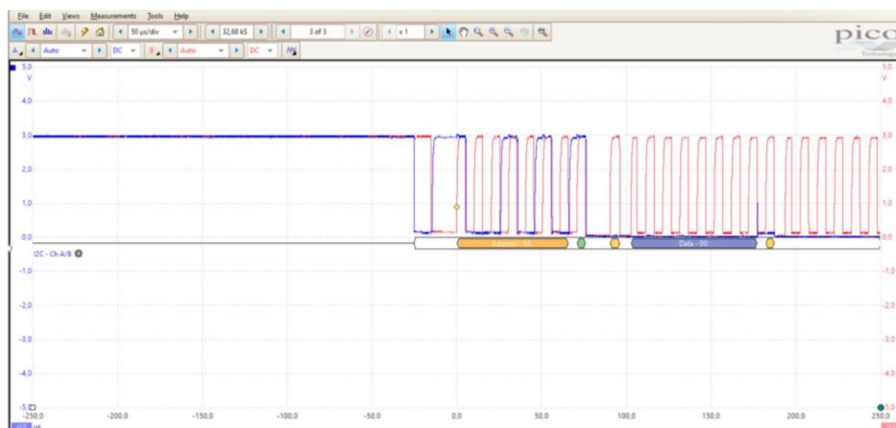
```
140
141         }
142         /* Switching to Low power from any */
143         else if (new_pm_state == PM_DEVICE_STATE_LOW_POWER) {
144             struct i2c_mux_data *data = dev->data;
145
146             /* Put the chip into sleep mode */
147             const struct i2c_mux_config *cfg = dev->config;
148             uint8_t mux_channel = 0;
149             uint16_t address = 0x70;
150
151             ret = i2c_write(data->i2c, &mux_channel, 1, address);
152             pm_device_put(data->power_device);
153             //dev->pm->state = new_pm_state;
154             //pm_device_put(data->power_device);
155
156             if (ret < 0){
157 /*              LOG_DBG("CTRL_MEAS write failed: %d",
158                     ret); */
159             }
160         }
```

Picture 11. A screen capture of VS code that shows an I2C write function fails and returns a -5.

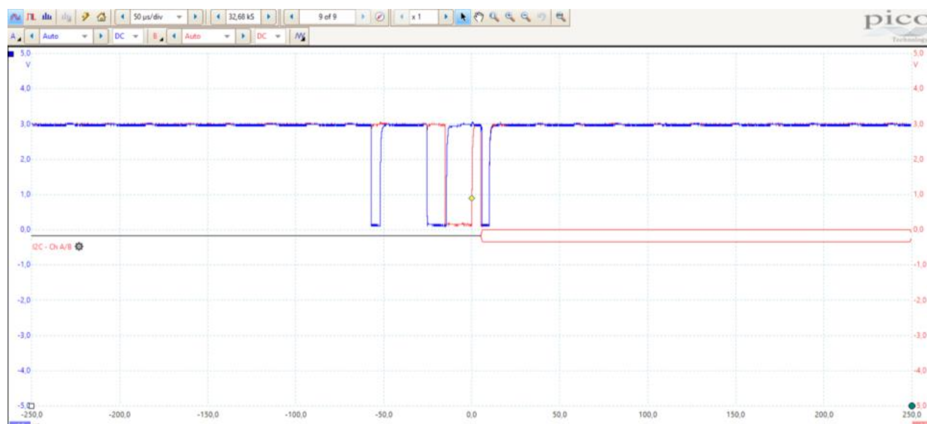
The job of the I2C write function shown in Picture 11 is to turn off the power line of channel zero, which is the power line for the BNO0x sensor when the sensor completes

an execution and is ready to enter a low power state. This explains the continuous spikes shown in Figure 15 in subsection 5.3.

An I2C write might fail due to several different reasons including, the attempt to write to the wrong address or to write the wrong data. One way to find out the cause for this failure to write is to prob the I2C SDA and SCL lines with an oscilloscope. For this particular write, the expected write sequence, which was also the result obtained from the write function before the system enters a low power state, is shown in Picture 12. Surprisingly, when the system enters a low power state, both the address and data were found to be completely corrupted as shown in Picture 13.



Picture 12. A screen capture from the Pico oscilloscope that illustrates the correct I2C write sequence obtained before the system enters a low power state. Where 0x40 is the address of the multiplexer and the data is zero.



Picture 13. A screen capture from the Pico oscilloscope that shows only two rising edges of the clock and corrupted data.

The corrupted data shown in Picture 13 is possibly caused by a software bug in Zephyr's power management framework or in the SoC's main driver. Alternatively, this could be caused by a hardware bug in the SoC, for which reason it was reported to the SoC manufacturer. The SoC manufacturer team were able to replicate the error and confirm that it is indeed a bug. However, no fix nor efficient workaround to this problem was found at the time this thesis was written. Further work is therefore needed to unravel this issue as well as other possible undiscovered design flaws. Additionally, in this work only three devices were configured to implement dynamic power management. Therefore, all device drivers in the system must be configured and added to the device runtime policy manager in order to achieve ultra-low power consumption with a variety of different applications.

7 CONCLUSION

This thesis explored hardware and software methods that helped enhance energy efficiency in a resource constrained embedded device. This device is under development and runs a real time operating system called Zephyr. The experimental results obtained from this work showed a dramatic drop in power consumption, with which the device can preserve battery charge for several months in a standby state, and up to roughly one moth of running a motion detection application.

This work was mainly focused on rectifying hardware design flaws as well as customizing device drivers with extended power management schemes. Such work requires a deep understanding of board layouts, circuit schematics, and component data sheets. To this end, only three device drivers were customized, and thus future work is needed to customize all other device drivers in the system. Additionally, possible hardware or software bugs might still exist and must be examined to achieve optimal power savings.

REFERENCES

Abdelmotalib, A. & Wu, Z., 2012. Power Management Techniques in Smartphones Operating Systems. *International Journal of Computer Science Issues,* 9(3), pp. 157-160.

Analog Devices, 2015. *Analog Devices.* [Online]. Available at: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7147.pdf. [Accessed 24 June 2021].

Angelini, C. & Wallossek, . I., 2014. *Intel Core i7-5960X, -5930K And -5820K CPU Review: Haswell-E Rises.* [Online]. Available at: https://www.tomshardware.com/reviews/intel-core-i7-5960x-haswell-e-cpu,3918-10.html
[Accessed 17 July 2021].

ARMLtd, 2019. *From one Arm to the next!.* [Online]. Available at: https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/armchip.html. [Accessed 19 July 2021].

Bigot , P. & Helm, M., 2021. *zephyrproject-rtos.* [Online]. Available at: https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/sensor/hmc5883l. [Accessed 25 July 2021].

Bolivar, M., 2021. *A deep dive into the Zephyr 2.5 (and 2.6) device model.* ZephyrIoT, Zephyr Project Developer summit.

Braza, J., 2020. *Circuit Basics.* [Online]. Available at: https://www.circuitbasics.com/pull-up-and-pull-down-resistors/. [Accessed 10 June 2021].

Ceolin, F., 2021. *Zephyr Power Management - 101.* s.l., The Linux Foundation project.

Chen, C., Novick, G. & Shimano, K., 2000. *RISC architecture "RISC VS CISC".* [Online]. Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/index.html. [Accessed 27 June 2021].

Chen, K.-H., 2016. *Power Management Techniques For Integrated Circuit Design.* Taiwan: John Wiley & Sons, Incorporated. ProQuest Ebook Central.

Clarysse, I., 2019. *Zephyr – An Operating System for IoT.* [Online]. Available at: https://www.zephyrproject.org/zephyr-an-operating-system-for-iot/. [Accessed 18 July 2021].

Dubois, C., 2018. *Who Really Owns Hardware? Property Rights vs Copyrights.* [Online]. Available at: https://www.allaboutcircuits.com/news/who-really-owns-hardware-property-rights-vs-copyrights/. [Accessed 3 September 2021].

Elvstam, A. & Nordahl, D., 2016. *Operating systems for resource constraint Internet of Things,* Malmö: Malmö University.

Farahani, S., 2008. Battery Life Analysis. In: *ZigBee Wireless Networks and Transceivers.* Newnes: Elsevier, pp. 207-224.

Gabelle, F., 2021. *Narrowband-IoT Power Saving Modes,* Oulu: Oulu University of Applied Sciences.

Grochowski, E. & Annavaram , M., 2006. *Energy per Instruction Trends in Intel Microprocessor.* [Online]. Available at: https://www.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf [Accessed 1 July 2021].

Gruener, W. & Miconi, C., 2018. *Intel Processors Over the Years.* [Online]. Available at: https://www.businessnewsdaily.com/10817-slideshow-intel-processors-over-the-years.html. [Accessed 27 June 2021].

IBM, 2012. *RISC Architecture.* [Online]. Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/risc/. [Accessed 27 June 2021].

Intel Corporation & Wind River Systems, 2021. *Devicetree versus Kconfig.* [Online]. Available at: https://docs.zephyrproject.org/latest/guides/dts/dt-vs-kconfig.html?highlight=kconfig. [Accessed 25 July 2021].

Intel Corp & Wysocki, R., 2016. *Device Power Management Basics.* [Online]. Available at: https://www.kernel.org/doc/html/v4.14/driver-api/pm/devices.html. [Accessed 15 July 2021].

Klasson, F. & Hecktor, Y., 2010. *Security in Embedded Systems,* Sweden: Linköpings University.

Liljasto, T., 2019. *Power Management in ARM Cortex M0+ with Real Time Operating System,* Helsinki: Metropolia University of Applied Sciences.

Lima, F., 2009. *Integration of Power Management Units onto the SoC.* Berlin: Springer.

Linux Foundation, 2020. *Zephyr API Reference.* [Online]. Available at: https://docs.zephyrproject.org/latest/reference/index.html. [Accessed 0.4.05 May 2021].

Malewski, M., Cowell, D. & Freear, S., 2018. Review of battery powered embedded systems design for mission-critical low-power applications. *International Journal of Electronics,* 105(6), pp. 893-909.

Mishin, K., 2017. *Low Power Firmware Design in Embedded Systems,* Helsinki: Metropolia University of Applied Sciences.

Mujtahid, M., ASofi, S. & Bashir, T., 2018. Comparative Advantages of System on Chips in Intelligent Traffic System. *International Journal of Robotic Engineering,* 3(1), pp. 2-9.

Noergaard, T., 2005. *Embedded Systems Architecture- A Comprehensive Guide for Engineers and Programmers.* Burlington: Elsevier.

Nordic Semiconductor, 2021. *nRF5340 DK - Development kit for the nRF5340, a dual-core Bluetooth 5.2 SoC supporting Bluetooth Low Energy, Bluetooth mesh, NFC, Thread and Zigbee.* [Online]. Available at: https://www.nordicsemi.com/Products/Development-hardware/nrf5340-dk. [Accessed 31 Julay 2021].

NXP, 2021. *NXP.* [Online]. Available at: https://www.nxp.com/docs/en/application-note/AN1902.pdf. [Accessed 06 June 2021].

Patch, K. & Smalley, E., 2001. *Computer Chips : Post-Moore's-Law Possibilities.* Boston: Technology Research News. Available from: ProQuest Ebook Central. [25 June 2021]..

Pelaez, A., 2021. *9 IoT Operating Systems To Use in 2021 [List & Comparison].* [Online]. Available at: https://ubidots.com/blog/iot-operating-systems/. [Accessed 18 July 2021].

Pinsheng Electronics, 2021. *PS Electronics.* [Online]. Available at: https://www.quick-pcba.com/pcb-news/ic-packaging-types.html. [Accessed 20 June 2021].

Remler, D., Das, S. & Jayanti, A., 2020. *Battery Technology.* USA: Harvard Kennedy School.

Richert, F., 2020. *Power and Beyond.* [Online]. Available at: https://www.power-and-beyond.com/introduction-to-electronic-compontents-active-vs-passive-components-a-893768/. [Accessed 19 June 2021].

Roser, M. & Ritchie , H., 2020. *Our World in Data.* [Online]. Available at: https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png. [Accessed 25 June 2021].

Saxena, S. & Akashe, S., 2015. *Design of Low Leakage Current Average Power CMOS Current Comparator using SVL technique.* Gwalior, India, IEEE.

Schlachter, F., 2013. *No Moore's Law for batteries.* [Online]. Available at: https://www.pnas.org/content/110/14/5273. [Accessed 9 July 2021].

Shalf, J., 2020. *Thefuture of computing beyond Moore's Law.* [Online]. Available at: https://www.researchgate.net/publication/338699741_The_future_of_computing_beyond_Moore's_Law. [Accessed 26 June 2021].

Shearer, F., 2008. *Power Management in Mobile Devices.* Texas: Elsevier Science & Technology.

Shepherd, P., 2002. *Integrated Circuit Design, Fabrication and Test.* New York: Palgrave Macmillan.

Shirriff, K., 2015. *Reverse engineering the ARM1, ancestor of the iPhone's processor.* [Online]. Available at: http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html. [Accessed 15 July 2021].

Simonović, M. & Saranovac, L., 2013. Power Management Implementation in FreeRTOS on LM3S3748. *Serbian Journal Of Electrical Engineering,* 10(1), pp. 199-208.

Sinha, P., 2008. *Microchip Technology Inc..* [Online]. Available at: https://www.microchip.com/stellent/groups/sitecomm_sg/documents/devicedoc/en542976.pdf. [Accessed 10 June 2021].

ST-Microelectronics, 2021. *LIS2DW - STMicroelectronics.* [Online]. Available at: https://www.st.com/en/mems-and-sensors/lis2dw.html. [Accessed 1 August 2021].

Sundriyal, V. & Sosonkina, M., 2017. Modeling of the CPU Frequency to Minimize Energy Consumption in Parallel Applications. *Sustainable Computing: Informatics and Systems,* 2(17).

Texas Instruments , 2013. *Texas Instruments Incorporated.* [Online]. Available at: https://www.ti.com/lit/ds/symlink/pca9535.pdf?ts=1624281472824&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FPCA9535%253FHQS%253DTI-null-null-octopart-df-pf-null-wwe. [Accessed 23 June 2021].

Tooley, M., 2006. *Electronic Circuits: Fundamentals and Applications.* Third ed. New(Jordan Hill, Oxford OX2 8DP, UK): Elsevier.

Tronicszone, 2020. *Power Management in Embedded Systems.* [Online]. Available at: https://www.tronicszone.com/blog/power-management-in-embedded-systems/. [Accessed 22 July 2021].

Vaddagiri, S., 2004. *Power Management in Linux-Based Systems.* [Online]. Available at: https://www.linuxjournal.com/article/6699. [Accessed 13 June 2021].

Walton, J., 2020. *Nvidia Unveils Its Next-Generation 7nm Ampere A100 GPU for Data Centers, and It's Absolutely Massive.* [Online]. Available at: https://www.tomshardware.com/news/nvidia-ampere-A100-gpu-7nm. [Accessed 25 June 2021].

WikiChip, 2021. *Semiconductor & Computer Engineering.* [Online]. Available at: https://en.wikichip.org/wiki/WikiChip. [Accessed 14 July 2021].

Wolf, W., 2006. *High-Performance Embedded Computing : Architectures, Applications, and Methodologies.* Burlington: Elsevier Science & Technology. Available from: ProQuest Ebook Central. [25 June 2021]..

Zhai, Y., 2018. *Handbook on Battery Energy Storage System.* Mandaluyong: Asian Development Bank.