Lauri Marjanen

Power management of embedded Linux systems

Metropolia University of Applied Sciences

Bachelor of Engineering

Smart Systems

Bachelor's Thesis

7 November 2022

Abstract

Author:	Lauri Marjanen
Title:	Power management of embedded Linux systems
Number of Pages:	32
Date:	7 November 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Smart Systems
Supervisors:	Joonas Möykkynen, Senior Software Designer
	Lauri Välimaa, Project Manager
	Antti Piironen, Principal Lecturer

Wapice Oy is releasing a new hardware revision of its embedded Linux-based WRM (Wapice remote management) platform. The new hardware design has implemented various power management features to expand its potential use cases to power-constrained environments. The objective of this thesis was to implement the latest hardware features into the device drivers and to create a remote configurable low-power mode for the WRM platform.

For this low-power mode, an additional software module for handling the system's low-power state was implemented into the system's remote management software, and various Linux drivers and kernel patches were created for the system to reduce power usage in sleep states and facilitate different system wake-up methods.

The document goes over the general power management features of modern embedded systems, the Linux kernel and its support of power management features and the implementation of the low-power mode for the WRM device.

Keywords: Linux, hardware drivers, power management

Tiivistelmä

Tekijä:	Lauri Marjanen
Otsikko:	Sulautetun Linux-järjestelmien virranhallinta
Sivumäärä:	32
Aika:	7.11.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine:	Älykkäät järjestelmät
Ohjaajat:	Vanhempi ohjelmistosuunnittelija Joonas Möykkynen Projektipäällikkö Lauri Välimaa Yliopettaja Antti Piironen

Wapice Oy julkaisee uuden laitteistoversion sulautetusta Linux-pohjaisesta WRM (Wapice remote management) -alustastaan. Uudessa laitteistosuunnittelussa on toteutettu erilaisia virranhallintaominaisuuksia, joilla laajennetaan sen potentiaalisia käyttötapauksia myös virrankulutukseltaan rajoitettuihin ympäristöihin. Tämän opinnäytetyön tavoitteena oli toteuttaa uusimmat laitteisto-ominaisuudet laiteen ajureihin ja luoda WRM-alustalle etäkonfiguroitavissa oleva virransäästötila.

Tätä virransäästötilaa varten järjestelmän etähallintaohjelmistoon lisättiin uusi ohjelmistomoduuli järjestelmän virransäästötilan käsittelyä varten. Uusia laitteistoajureita luotiin ja olemassa olevia ajureita muokattiin saadakseen virrankulutusta vähennettyä lepotiloissa ja lisätäkseen erilaisia tapoja järjestelmän herätykseen.

Asiakirjassa käsitellään nykyaikaisten sulautettujen järjestelmien yleisiä virranhallintaominaisuuksia, Linuxia ja sen tukea virranhallintaominaisuuksille sekä WRM-laitteen virransäästötilan toteutusta.

Avainsanat: Linux, laitteisto ajurit, virranhallinta.

Contents

List of Abbreviations

MMP: Memory-mapped peripheral. A device that is interfaced by accessing specific memory regions.

MMC: MultiMedia Card. Communication bus for communicating with SD cards or other external peripherals.

RISC: Reduced Instruction Set Computer. A computer design philosophy designed to simplify the CPU instructions given to the computer.

ARM: Advanced RISC machine. Low-power CPU core primarily used for mobile and embedded devices.

I2C: Inter-Integrated Circuit. Communication bus commonly used for embedded low-speed peripherals.

LTE: Long Term Evolution. Standard for wireless broadband communications.

GNSS: Global Navigation Satellite System.

WLAN: Wireless Local Area Network.

CAN: Controller Area Network. CAN is a message-based communication protocol that allows multiple devices to communicate on the same line.

GPIO: General purpose input-output. Interface for configuring and controlling input-output pins.

USB: Universal Serial Bus. An industry standard for cables, connectors, and protocols for interfacing between computers and peripherals.

PCI: Peripheral Bus Interconnect. Communication bus for adding computer components to the system.

UART: Universal Asynchronous Receiver Transmitter. A computer hardware device used for asynchronous serial communication between devices.

RS-232: Recommended standard 232. Electrical communication standard used for serial communication.

RS-485: Recommended standard 485. Electrical communication standard used for serial communication.

MDIO: Management Data Input/Output. Communication bus used in data-link layer for ethernet standard.

MPU: Microprocessor unit.

1 Introduction

Power consumption places limitations on most computational system designs either by increasing operating costs, drawing more power than available, or generating more heat than the system can dissipate. Power management needs to be considered in all systems of various sizes, from data centres limited by heat dissipation to mobile devices limited by battery capacity. [1, 2]

Modern operating systems, such as Linux offer their users various power management features for reducing the system's power consumption. To facilitate the development of these features, the operating system comes with readymade tools and software frameworks for managing the device's power usage.

The idea of this thesis was to implement power management features for a Linux-based remote management device. Development of the feature required hardware-specific changes to be implemented to the device's Linux device drivers and the addition of user-configurable software responsible for managing system sleep states. The feature was implemented for Wapice Oy's WRM247LTE device.

This thesis is divided into four parts. Chapter 2 reports on sources of power consumption, different power management techniques and their trade-offs. Chapter 3 introduces the Linux kernel, its driver interface and power management features and how Linux devices can be configured through the device tree. Chapter 4 will cover the target device, the implementation of hardware-specific power management features for the Linux kernel and the addition of the low-power mode for the device's remote management software. Finally, chapter 5 will go over the results of the implemented feature.

2 Power management

Power consumption is one of the leading design constraints of an embedded system. There is always a need to reduce power usage to meet the system's power-constrains or to decrease electricity usage. Excessive power usage can reduce battery-powered systems working time while making some bus-powered devices completely unviable. A mobile phone with a 1-hour operating time or a USB mouse that requires a separate power source would be highly unpopular. The heat generated by the system's electricity consumption can also introduce increased requirements from a system's cooling solution, increasing the complexity and cost of the device. [3]

2.1 Circuit power management

The software has been taking increased responsibility for power management in modern embedded systems. While optimizing the power usage on the system's various silicon components, such as CPU or GPU, would produce the highest power savings. The system's software should also be aware of unnecessary power usage caused by the device's power islands and GPIO pins.

Power Island is a hardware design concept for grouping power-consuming devices into separate blocks. Primarily used in CPU and system-on-chip designs, the power islands can be individually powered down without impacting the components outside of the islands. Power islands allow the system to reduce the device's static power usage by disconnecting the power from system's various silicon blocks. [4]

The purpose of the GPIO system is to provide a software interface for configuring and controlling a physical pin. Both input and output pins can be in a low, high, and floating state. The low state equalling pin voltage level equal to circuit ground, high condition equal to the circuit voltage source, and floating state is unknown. Pins configured as outputs will draw current when the pin is high. When pins are configured as inputs, a pull-up or pull-down resistor must

be added to read the pin's logic level. The input pins consume power when the logic level is opposite of what it is pulled to, and current flows to the ground through the resistor.[5] Below is an illustration of GPIO pins configured as inputs with both pull-up and pull-down resistor configurations.

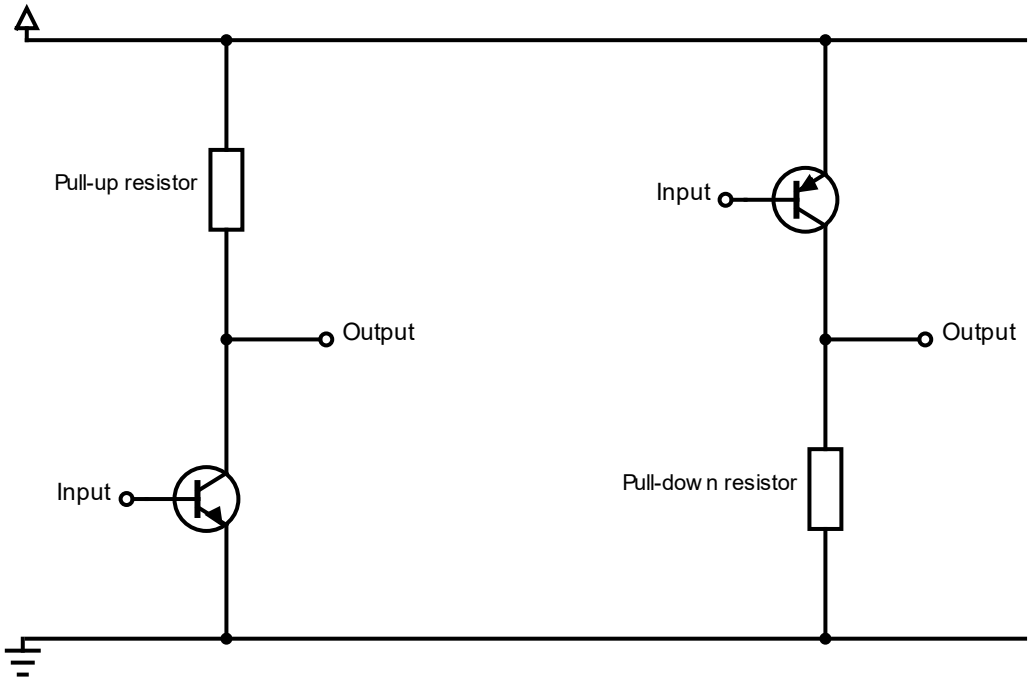

Figure 1. Image of input pins with pull-up and pull-down resistor configured

Figure 1 illustrates both pull-up and pull-down resistor configurations of a GPIO input pin. Power is consumed when the transistor is activated, and current flows from the voltage source to the ground.

During system sleep states, pins can have small current leaks depending on what state the pin is left in. GPIO pins should be configured into a low-power state when not needed. Methods for saving power on gpio pins vary with hardware platform and manufacturer recommendations. In situations where the manufacturer has not implemented a low power state for gpio pins, the output-low configuration should be used. Communication busses such as I2C and MMC utilize more power-consuming pin configurations that use the pins floating

state, such as push-pull and open-drain. Making the pins for communication buses good candidates for low power mode during sleep. [6]

2.2 CPU power management

Modern CPUs are usually the most complex and expensive parts of the system, as they are designed to operate continuously with varying power loads for their entire lifetime. Power consumption of a CPU can be categorised into static and dynamic. Static power consumption occurs when electricity is supplied to silicon blocks such as CPU core or cache. Static power loss is constant and arises from keeping the circuit in a known logic state due to current leaking through the silicon transistors. In contrast, the dynamic power consumption comes from changes in transistor states and varies based on the work.[7] Static power consumption has increased exponentially with transistor size reduction, while dynamic power usage has increased linearly. Figure 2 illustrates the dynamic and static power consumption ratio depending on the processor transistor size.

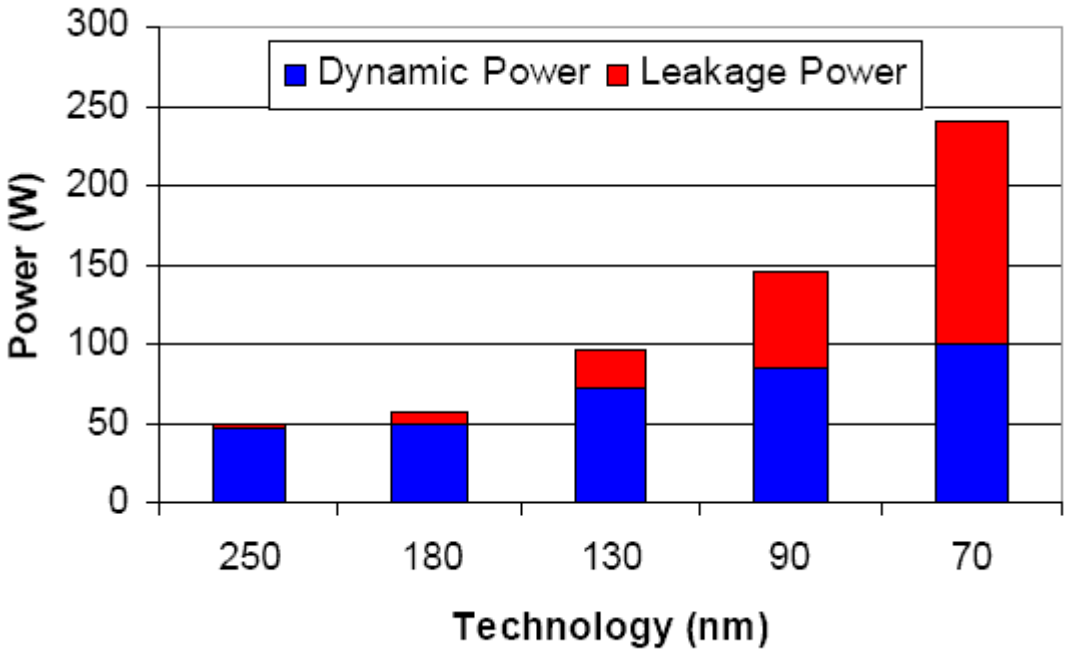

Figure 2. Dynamic and static(leakage) power consumption of different manufacturing nodes of Intel processors. [8]

The increasing ratio of static power usage in the CPU is illustrated in figure 2. Highlighting the importance of powering down unused parts of the circuitry, as static power usage serves no purpose for CPU usage.

It is improbable that a CPU is constantly fully utilized, so CPU designers have implemented power management features such as P- & C-states on Intel x86 or dynamic & static pm on the ARM architecture. These states reduce power usage by deactivating parts of the CPU cores or lowering their frequency. Management of these power states is left to the systems software, which will then use some algorithm to estimate what cores it can disable or how much it can adjust the CPU clock. [9]

Optimizing CPU power consumption by disabling idle CPU cores is desirable, especially on an embedded system. However, this comes with the downside of increased access latency for the suspended resource. As the system needs to perform a reinitialization or a resume sequence for the inactive resource every time it is required, the system can consume even more power than average due to work being done to bring resources back into use. Overly aggressive power management can cause possible power savings to be negated as the system spends time and electricity to transition states. In most instances lower the power consumption of a state, the higher the state transition latency and power consumption. [10] Depending on the device, the CPU can have any number of low-power states that optimize power usage to varying effectiveness. Figure 3 illustrates a simple three-state power machine for an older processor with Run, Idle and Sleep states.

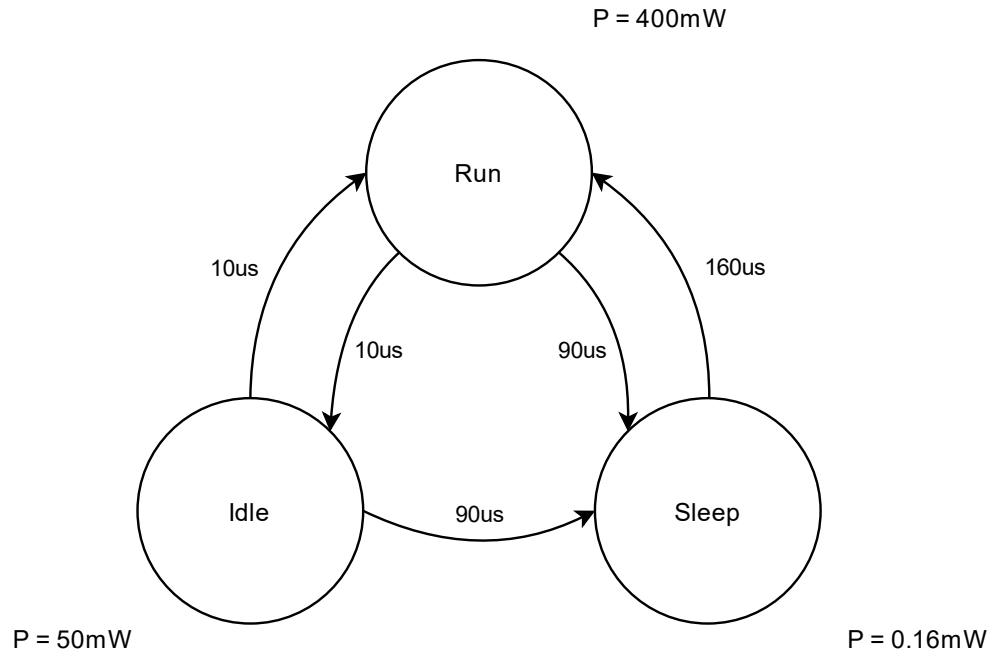

Figure 3. A simple power state machine for strongARM SA-1100 CPU. [11]

Figure 3 illustrates a three-state power machine. Run state is the normal operating mode of the CPU, with power consumption at the highest. The idle state offers moderate power savings and fast resume time by disabling the CPU clock and stopping the fetch decode execute cycle. Sleep state sets the CPU into low power mode, powering off most of the CPU's parts and requiring a costly resume sequence. Transitioning from an Idle or Sleep state to a Run state requires a wake-up signal as Idle and Sleep states pause logic execution. Returning from the Sleep to Run state takes the longest time because the chip must re-initialize more components before resuming regular operation.

3 Linux operating system

Using Linux as an operating system for an embedded system comes with all the benefits of using a modern Unix system, such as thread scheduler, networking stack, and memory management. These features can save a massive amount of overall development time but require a longer system setup time and increased knowledge requirements from the developer. Most CPUs that support embedded Linux come provided with readymade configurations necessary to boot the device and operate on-chip peripherals. Increased setup times come from creating custom devices, as added or modified system features must be configured correctly. Adding peripherals to the system requires a device tree entry where the device can be found or for the device to be enumerated through a communication bus such as USB or PCI. [12]

Linux operating system segregates its virtual memory into user space and kernel space. Processes running in kernel space have more direct access to systems hardware and core functionality, whereas system applications run in user space. Such a split is primarily meant to increase system security and stability by protecting the operating system from malicious or errant software installed by the user. [13]

Linux kernel facilitates interactions between system hardware and software, requiring all hardware access to be done through the abstractions provided by kernel syscalls. Figure 4 illustrates hardware access by the user program.

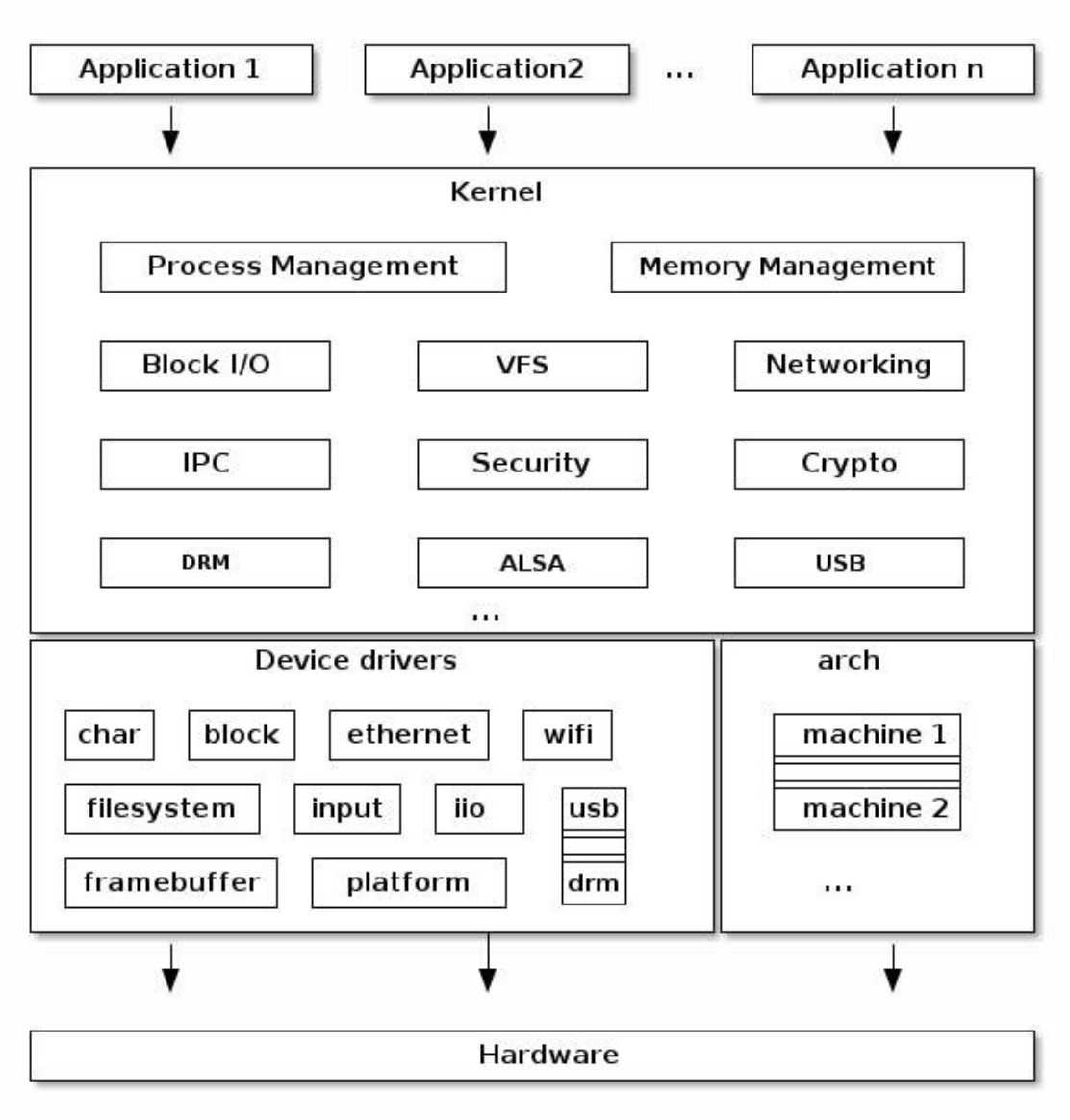

Figure 4. Linux syscall interface for interacting with hardware. [14]

Figure 4 shows an application accessing hardware resources through a Linux syscall. Processes from user space perform operations, such as sending data packets to the internet. The Linux kernel is responsible for forwarding the request to hardware through the device driver while ensuring that multiple programs do not interfere with each other.

3.1 Linux kernel drivers

In software development, a device driver is responsible for providing the system with an interface for a specific hardware device or another low-level service. Linux is a monolithic kernel, meaning its drivers are loaded and unloaded as modules by the system during runtime. When a driver is loaded, the driver's code is added as a part of the Linux operating system, and the functionality provided by the driver is added to the system. [15]

The process of loading a driver into the kernel varies based on the driver and hardware type. Generally, the process involves setting up the system for communicating with the device, discovering the target device, and finally configuring the target device for operation. [16]

With a modular approach to device drivers, it is easy to create dependency links between system components. Device drivers are linked based on their dependencies on other devices; that is, a driver for a communication bus is always loaded before and unloaded after the consumer device. The driver generates these device links based on the device-tree config or enumeration of the child device. [17] For example, the ds2482 device provides a 1-wire peripheral but communicates through the I2C bus. When a program requires access to the 1-wire temperature sensor, the program must load the drivers for I2C, ds2482 and 1-wire. Linux kernel would then initialize the system's I2C bus to access the ds2482 device, then use the I2C driver to probe for the ds2482 device and finally provide a 1-wire resource to the system. With this approach, the user program receives a 1-wire temperature sensor and does not need to concern itself with the communication method.

3.2 Device tree

A device tree is a simple data structure used by the Linux kernel to describe system hardware. Linux initially supported the device tree for PowerPC and SPARK platforms; support was later expanded for ARM and x86 platforms.

The primary purpose of the device tree is to separate the hardware configuration from the overall system software, removing the need for hardcoding hardware configuration to the Linux kernel source code. The device tree is compiled into a binary and passed to the Linux kernel at boot time. Informing the Linux kernel what platform operating system is running on, attached devices, and how hardware is configured. [18] Listing 1 depicts an example of device tree configuration.

```
…
i2c@f01c00 {
      compatible = “atmel,at91sam9x5-i2c”;
      pinctr-names = “default”;
      pinctr-0 = <&pinctr_i2c2>;
      …
      ds2482@18 {
            compatible = “maxim,ds2482”;
            status = “okay”;
            …
      }
}
```

Listing 1. A snippet of the device tree configuration for the ds2482 sensor.

An example in Listing 1 shows a device part of the tree for the ds2482 device attached to the I2C bus. Device tree nodes contain the “compatible” node, configuring what driver is used for the node. Configuration of the I2C node also includes the child device and properties related to hardware configuration, such as a link to the gpio pin node used for the communication bus. The Pinctr node is defined somewhere else in the device tree.

A device tree is structured like an acyclic graph with an arbitrary amount of named nodes containing data or links to other nodes. The root of the device tree generally contains devices connected to the CPU bus, after which the tree will spread out into subcomponents, busses, external peripherals, etc. Below is a figure depicting the organization of a device tree.

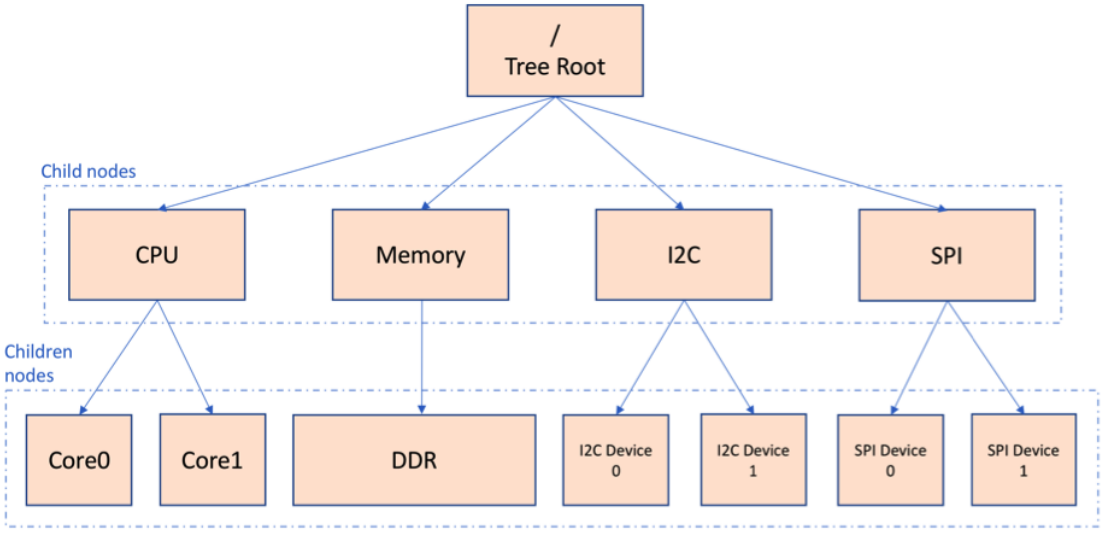

Figure 5. A simple device tree structure. [19]

As depicted in figure 5, the device tree structure spreads into child nodes. When loading a new device driver into the kernel, the Linux kernel will use device tree configuration to discover and configure all devices supported by the driver.

3.3 Power management in Linux kernel

The Linux community has implemented both suspend (static power management) and runtime (dynamic power management) power management for its driver framework. For a device driver to utilise some form of power management, it must have implemented the power management callback functions in its device driver. Drivers can have support for up to four different callback function pairs to improve the device driver's software control during the system's entering and leaving of sleep stages. The function pairs will be called during various system suspend and resume stages. [20] Listing 2 shows two callback function pairs for two system sleep stages and callback functions for drivers' runtime power management.

```
struct dev_pm_ops {
      int (*suspend)(struct device *dev);
      int (*resume)(struct device *dev);
      int (*suspend_late)(struct device *dev);
      int (*resume_early)(struct device *dev);
      …
      int (*runtime_suspend)(struct device *dev);
      int (*runtime_resume)(struct device *dev);
      int (*runtime_idle)(struct device *dev);
      ...
  };
```

Listing 2. Power management callback function for Linux drivers.

Above is a snippet of a Linux driver component responsible for the driver's power management. The kernel invokes these callback functions during the different system suspend and resume stages. Callback functions with runtime prefixes are used for managing device power during runtime.

Linux operating system supports two power management strategies: working-state and sleep-state. The working-state power management model allows the operating system to reduce power consumption by deactivating idle hardware components or lowering system performance. In most scenarios, only some system parts perform work at a given moment. Thus, allowing the operating system to save power by turning off or adjusting the operation of its inactive components. When the operating system suspects that the device is not in active use, it will invoke the device driver's idle check callback function and perform actions based on the result of the check. The effectiveness of this approach varies based on device type, driver implementation, and the power island device belongs to.

On the other hand, sleep-state power management is more of a system-wide approach. When the user initialises the system suspension, all processes are frozen, loaded drivers will undergo their respective suspend sequence, and finally, the system enters a state of reduced activity. When a program refuses to freeze or a driver generates an error during system suspend operation. The operation is aborted, and the system is prevented from falling asleep. [21] Below is a sequence diagram of a Linux system's suspend and resume process.

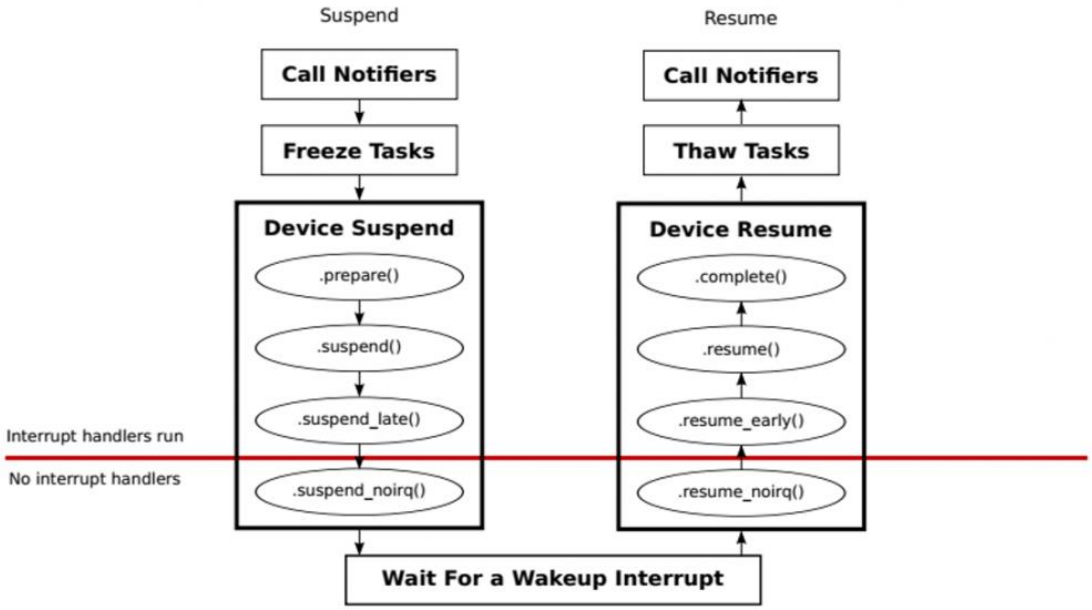

Figure 6. Linux suspend-resume process for idle state. [22]

Figure 6 shows a sequence for suspending and resuming the system from an idle state. When the suspend process is initialized, the kernel begins freezing system processes and executing power management callback functions for every driver. Any failure during the suspend sequence will abort the suspend, while a failure in the resume sequence will cause the driver to generate an error or process to terminate.

Linux supports four suspend states: idle, standby, ram and disk. All these states halt system operation and wait for a wake-up event. What differentiates them is the amount of power saved, wake-up latencies, and hardware requirements. The Idle state offers the lowest power saving and the fastest wake-up time. Whereas standby, ram and disk states conserve an increasing amount of energy by disabling more and more hardware components. The idle state is implemented purely in software, saving power by freezing user processes and setting I/O devices into power-saving mode. Standby state power saving is more aggressive, disabling CPU components in addition to all idle features. Ram and disk states put everything in the system into a low-power state, saving the entire system state either to ram or non-volatile storage. [23] The

appropriate system suspend state should be decided based on the duration of sleep, wake-up time and urgency of the task system wakes up to perform.

Some hardware devices may want to be able to wake the system from sleep. For this purpose, the power management system of the Linux kernel has added the wakeup framework for its power management framework. Drivers can register themselves as wake-up capable to indicate to the operating systems about its features. The operating system tracks wake-up capable devices, which can be viewed and configured from the user space. If a device wake-up has been enabled, the kernel informs the driver about its configuration. This method exists as a convenient way to manage the wake-up methods from the system’s sleep states, and it is left to the driver and the hardware to wake the system from sleep.

4 Implementation

Implementation of low-power features for the WRM247LTE device consisted of two parts, adding the device-specific power management and wake-up features to the Linux device drivers, and adding a user-configurable low-power mode for the system's remote management software. Linux-specific features were implemented using device tree configurations and Linux drivers. The control of the system's low-power mode was implemented as an additional software module for systems C++-based remote management software.

WRM247LTE is an upcoming hardware revision of Wapice Oy's WRM (Wapice Remote Management) platform. The WRM devices are primarily sold as a part of the Wapice IoT-Ticket remote management system but can be decoupled from the ecosystem and operated independently. System's remote management software, TerminalApp, handles the system's communication with the remote management server. TerminalApp loads required software modules that manage the configured system components. For example, the CAN module is responsible for collecting and sending data over the systems CAN communication bus. TerminalApp then relays the data from configured modules to the server and vice versa.

WRM247LTE supports many wired and wireless communication protocols, such as GPS, WLAN, cellular, USB, CAN, RS-232, RS-485 and ethernet. The upcoming WRM247LTE device has additional support for 4G cellular network and additional power-saving features. Below is an illustration of a previous-generation WRM device.

Figure 7. Image of input pins with pull-up and pull-down resistor configured

Figure 7 shows the WRM247PLUS devices case and various connectors. WRM247LTE device has an additional antenna port for the LTE functionality but otherwise retains the same base functionality and connectors.

During regular operation, WRM is either waiting for, or processing received data. This causes the device to consume more power than necessary if the device is not expected to perform any work at the given moment, making the power consumption of WRM prohibitively expensive in power-constrained environments. For this purpose, the new hardware has implemented power control for all the system’s hardware modules.

The large amount of communication protocols comes at an increased power cost, as additional hardware modules are required for each feature. Every

module has a different power-consumption footprint and method for reducing power usage. For the low-power mode of WRM247LTE, it was decided to power down the components as it is unnecessary to keep the system's various hardware modules powered during sleep. Table 1 lists the current draw of some system modules, as reported by module datasheets.

Component	Running(mA)	Idle(mA)	Sleep(mA)
LTE Modem	210.00	28.00	3.00
WLAN	212.00	0.08	0.08
Bluetooth	34.00	N/A	N/A
Ethernet1 transceiver	47.00	20.00	4.00
Ethernet2 transceiver	62.00	61.00	1.00
RS-232 line driver	1.00	0.30	N/A

Table 1. Components datasheets advertised current draw in milliamperes

Table 1 illustrates the current draw of system chips in different states. The actual power consumption value is between idle and running, as components will enter an idle state when possible. While an individual component's power usage in the sleep state is minor, combined power usage is significant.

4.1 Radio device's power management

For the WLAN & Bluetooth, WRM247LTE utilizes a combo module containing both features on the same chip. Bluetooth and WLAN only needed minor device tree configuration to implement the desired power management solution. Both drivers handle the power management of the device perfectly. When going to sleep, drivers turn off the hardware. After waking up, the WLAN & Bluetooth drivers restore the previous operation state.

WRM247LTE device's modem device provides both cellular LTE and GPS functionality for the system. No Linux driver existed for the modem device as it is configured and interacted with by the system through the USB protocol.

However, a driver was created to consolidate the start-up and shutdown sequence of the modem, as TerminalApp's various modules need to interact with the modem independently.

The modem driver required implementing power management from scratch, as it was not considered during its initial development. The functionality for power cycling the modem during system state transition was added for the driver, but the modem would lose its configuration during sleep. While it could have been possible to restore the modem's previous operation from the driver, it was deemed suboptimal to communicate with a USB device from kernel space. The issue was solved by having the TerminalApp software reconfigure the modem after the system had left the sleep state. An alternative approach would have involved setting the modem into low-power mode instead of powering down the device, thus preserving the device's configuration but increasing power usage in sleep. Below is a plantUML diagram of the power sequence for the modem driver.

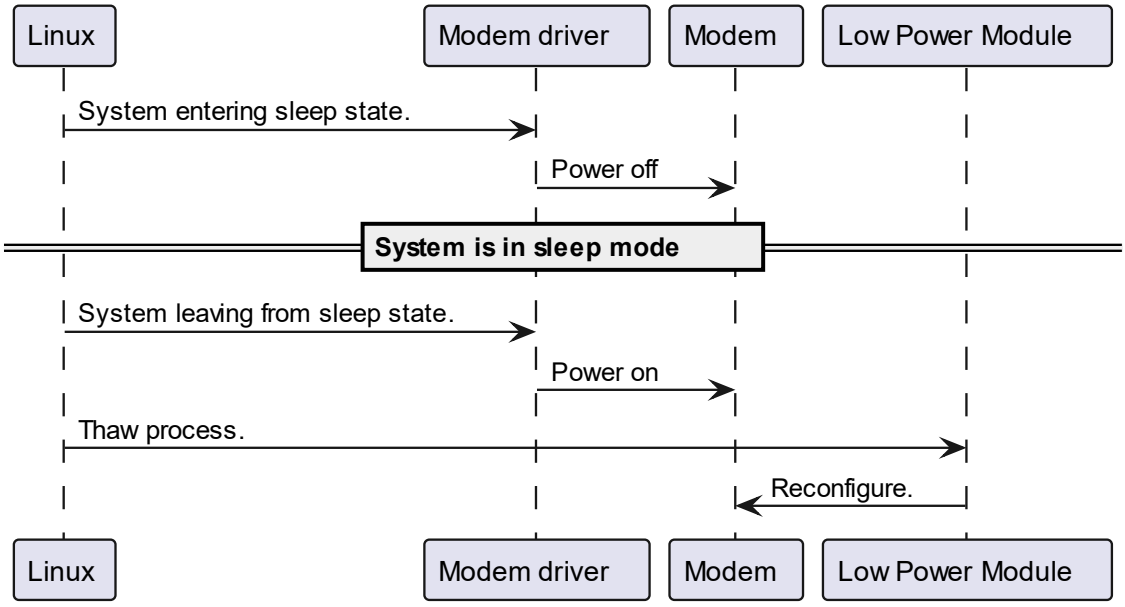

Figure 8. Modem devises power sequence during system sleep and wakeup process.

Figure 8 shows the process of re-initializing the modem after the system wakeup. As the modem configuration is lost during sleep and the modem requires reconfiguring from the user space, a method for re-initializing the modem was implemented into the low-power module.

4.2 GPIO power management

During testing of the device's power consumption, it was noticed that specific GPIO pins were leaking power while the system was asleep. Current leakage was mainly observed on GPIO pins for the I2C and MMC communication busses due to pins being left in a floating state. Output pins could also cause current draw if they are in a high state, supplying current to a powered-down component. For these reasons, it was necessary to set most GPIO pins to a low power state during sleep. For the MPU of WRM247LTE, the manufacturer did not specify a low-power state for individual pins. Various GPIO configurations were tried, and a simple GPIO output low state gave the best results.

Initially, three different solutions were considered for the problem. The first method was to implement a kernel patch for each communication bus driver to set the pins to the desired state. This method has its downside, as each kernel patch needs to be re-implemented when upgrading the Linux version. The second method to be considered was creating a single hook patch in the Linux kernel's power management functionality to set the pins into the desired state and back to the previous state. This would reduce the number of kernel patches to maintain to one. Finally, creating an entire driver for managing the system GPIO pin states in sleep was also considered. This would work by reconfiguring the pins in the latest possible suspend stage. The driver method would be the most portable solution, as no kernel patches would be needed, and the risk of future Linux versions breaking the driver was minimal. After investigating how other drivers handle such an issue, it was noticed that the Linux kernel's pinctr framework already offers a solution to the problem. And the first kernel patch method was chosen to keep the solution consistent across device drivers.

Linux kernel uses the pinctr framework to set the GPIO pin's settings based on the information provided by the device tree. Pins are set according to the configuration into input or output. Additionally, various other settings can be configured for the pin, such as pull-up or pull-down resistor, debounce etc. Linux drivers can change the pin's state according to their needs, such as active, idle and sleep. This approach is helpful for GPIO pin configurations that leak current, such as open-drain on I2C or push-pull on MMC. Listing 3 below shows a pinctr device tree config for I2C that supports swapping pins from active to sleep state.

```
i2c_pinctrl_state_active: i2c_pinctr_active {
     atmel,pins =
          <AT91_PIOA 18 AT91_PERIPH_B AT91_PINCTRL_NONE
           AT91_PIOA 19 AT91_PERIPH_B AT91_PINCTRL_NONE>;
};
i2c_pinctrl_state_sleep: i2c_pinctr_sleep {
     atmel,pins =
          <AT91_PIOA 18 AT91_PERIPH_B AT91_PINCTR_SLEEP_STATE
           AT91_PIOA 19 AT91_PERIPH_B AT91_PINCTR_SLEEP_STATE>;
};
…
i2c : i2c@f8018000 {
     Compatible = "atmel,at91sam9x5-i2c";
     …
     pinctrl-names = "active", "sleep";
     pinctrl-0 = <&pinctr_state_active>;
     pinctrl-1 = <&pinctr_state_sleep>;
};
```

Listing 3. Device tree config for I2C pinctr sleep states.

Listing 3 shows the used pinctr configuration of the I2C module with two states, active and sleep. This configuration allowed the I2C driver can set pins into a sleep state when needed.

The drivers for the I2C and MMC communication busses already supported changing pin states during systems suspend operation, and only an additional device-tree configuration was needed. A similar method for changing the pinctr pin states was implemented using kernel patches for the rest of the communication buses. Measured power savings from setting I2C and MMC powerlines into low-power state amounted to 6mW from the I2C bus and 101mW from the MMC line. Power savings from other busses was negligible.

4.3 Low-power groups

On WRM247LTE, most hardware modules have a shutdown pin that can be asserted to power down the device or to disconnect them from the power bus. Due to the limited availability of GPIO pins on the system CPU, the power control pins of some components were grouped into a single line. Two low-power groups exist, one for physical ethernet circuits and another for UART to RS-232 line drivers. This caused issues as two different drivers would control the power source of multiple devices, and the second device would always undergo power loss before its device driver was aware of the hardware change.

The Linux kernel's voltage regulator framework provided a ready-made solution for such a problem. With a regulator, multiple device drivers can share a power source. The operating system then ensures that no other device is powered off because a single device shares a power source with another. Listing 4 shows an example of a fixed regulator device tree configuration.

```
fixed-regulator: fixed-regulator@1 {
  compatible = "regulator-fixed";
  regulator-name = "fixed-regulator";
  gpio = <&gpio 16 0>;
  startup-delay-us = <70000>;
  enable-active-high;
};
```

Listing 4. Example of a regulator that controls a single GPIO line.

Listing 4 is a snippet from a regulator that controls a single gpio line. In this case, the regulator controls the usage of GPIO line 16. The start-up delay time parameter can be used to ensure that the hardware module has enough time to power on.

The serial and ethernet drivers did not support such a low-power group design, and a kernel patch had to be implemented to add regulator support for both drivers. The process involved creating a regulator for each low-power group in the device tree and linking it to serial and Ethernet nodes. Changing the Linux kernel was accomplished by creating a patch for the source code.

Implementing the regulator for the ethernet transceiver hardware modules was the most time-consuming aspect of the work. A large amount of time was spent investigating an issue where about 20% of the system's suspend and resume cycles caused the ethernet to stop working, requiring the system to be power cycled so that regular ethernet operation could be resumed.

Systems both ethernet transceivers, the components responsible for transmitting and receiving data through the system's ethernet ports. Would sometimes fail to power on after the system was back from sleep. Various possible solutions were investigated without results: changes to transceiver initialization sequences, resetting the Linux ethernet and transceiver drivers, and power cycling the low power group again after failure was detected. Accessing the transceiver registers to investigate the device configuration before and after sleep provided a small clue. None of the registers could be read or written to, so a software solution was unlikely to be found. The problem ended up being in a faulty WRM247LTE development unit that leaked a small amount of current into a powered-down ethernet transceiver, causing it to lock up during the power-up sequence and requiring the complete system power disconnect to resume regular operation.

The power of the low power group for the device's serial ports was handled inside the Linux serial driver. In contrast, the low power group for the ethernet transceivers had to be addressed in the parent bus device of the transceiver driver. The process of loading and configuring the correct ethernet transceiver driver involves the following:

- Loading systems MAC driver.
- Discovering ethernet transceiver device.
- Loading the appropriate device driver.

As the MDIO bus handles the discovery of the transceiver device, the transceiver needs to be powered on before the MDIO bus can be utilized. Additionally, transceiver drivers are derived from the ethernet-PHY driver. That

is incapable of implementing systemwide power callback functions making the MAC driver the best choice.

4.4 WRM low-power mode

An additional software module had to be implemented for the TerminalApp application. This low-power module would be responsible for configuring the wake-up methods for the system and setting the system into sleep mode when desired. The defined use case for the low-power module was:

- Device enters the lowest power-consuming state.
- Device wakes up to handle a configured hardware signal, then continues sleeping.

Due to the generic purpose of the low-power module, predicting when the system should go back to sleep was difficult. Handling a specific wake-up signal and sleeping would need to be tailor-made for the customer use case, so a simple timer was used instead. Additionally, the maximum sleep length had to be considered if the user wanted to change the configuration of WRM247LTE. As it would be undesirable if the system sleeps forever due to the wake-up signal never being received or wrongly configured, requiring the device to be manually rebooted.

The user interface for low-power mode can be used to customize the module’s various operational parameters, allowing for fine-tuning based on the customer’s expected usage of the WRM24LTE device. Below is an illustration of the configuration interface for low-power mode.

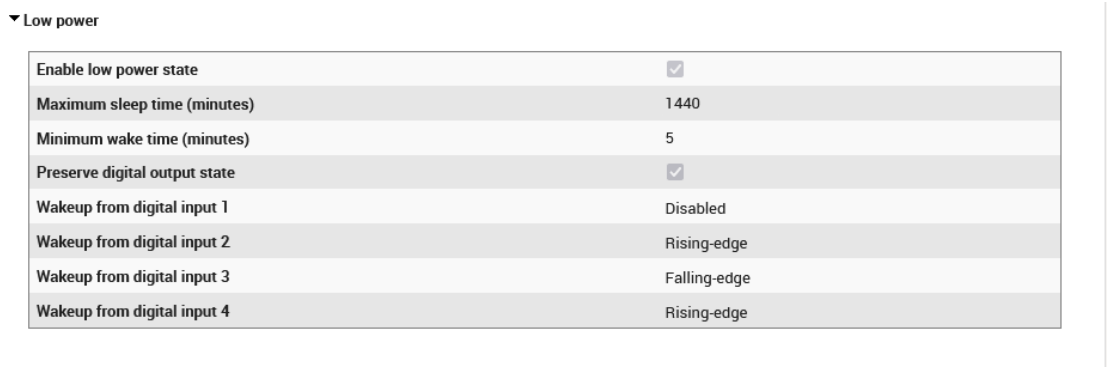

Figure 9. Configuration interface for low-power mode

Figure 9 shows the user interface used for configuring a WRM247LTE device's low-power mode. Users can configure the length of the system's sleep and wake-up time in minutes. Additionally, the behaviour of digital output pins in sleep and individual wake-up sources can be configured.

When the system boots up or receives a new configuration, TerminalApp loads all modules configured by the remote management software. The low-power module starts by creating a thread that tracks the system's wake-up time, then periodically checking if the system sleep sequence needs to be activated. Periodic checks are required to avoid systems watchdog module terminating long-sleeping threads. The module begins with a wait timer to prevent the system from falling asleep instantly after the device boot or configuration update. Below is a plantUML diagram of the low-power module initialization sequence.

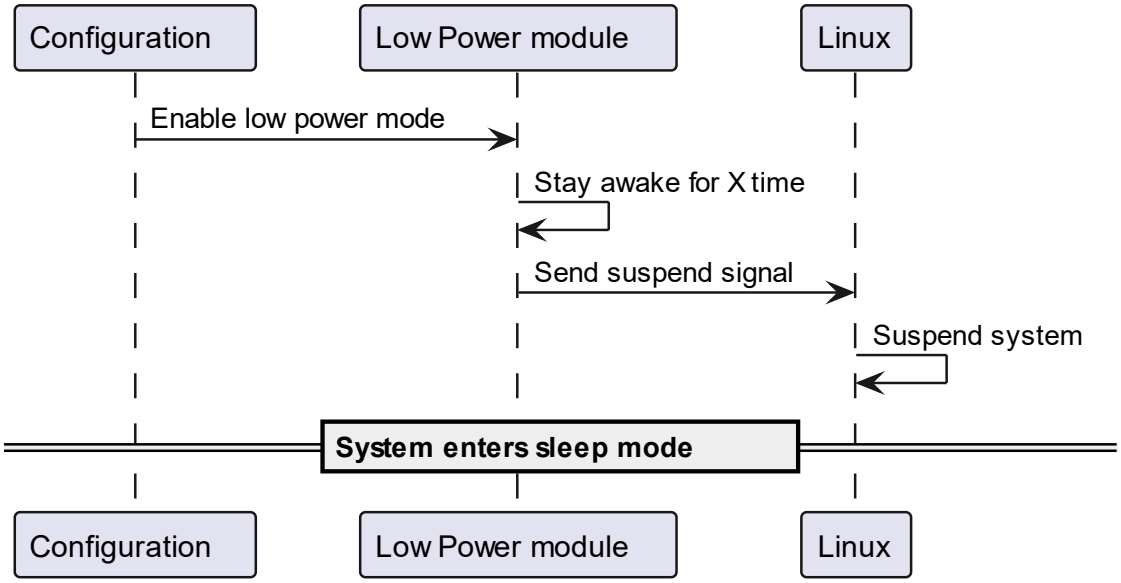

Figure 10. Initial entering of the sleep mode.

Figure 10 shows the initialization process of the low-power module. TerminalApp initializes a thread for the low-power module. To avoid the system instantaneously falling asleep when the module loads, the module is started in the stay awake phase.

To facilitate the setting system to sleep for the configured value, the low power module sets the system to sleep utilizing the rtcwake Linux program. When the module executes the rtcwake command using the system() C function, the system will sleep and wake up during the execution of the system() function call. Below is a code snippet that sets the system to sleep using the rtcwake program.

```
void LowPowerModuleThread(){
      …
      printf("System entering s2ram state\n");
      system("rtcwake -m mem -s 60");
      printf("System left s2ram state\n");
      …
}
```

Listing 5. A code snippet that sets the system to an s2ram state for 60 seconds

The code shown in listing 5 sets the system into a low power suspend to ram state. The first print statement is executed before sleep, and the second print

after the system has woken up. Low-power module utilizes a similar code to track the system entering and leaving sleep.

4.4.1 System Wake-up methods

The low-power module supports waking the system up from the system clock and the device's digital input pins. Waking up from the system clock is automatically set by the rtcwake program that is used to suspend the system. Waking up from the digital input pins required creating a separate Linux driver that handles the configuration of the GPIO wake-up. Below is a plantUML diagram of the system waking up to a hardware signal.

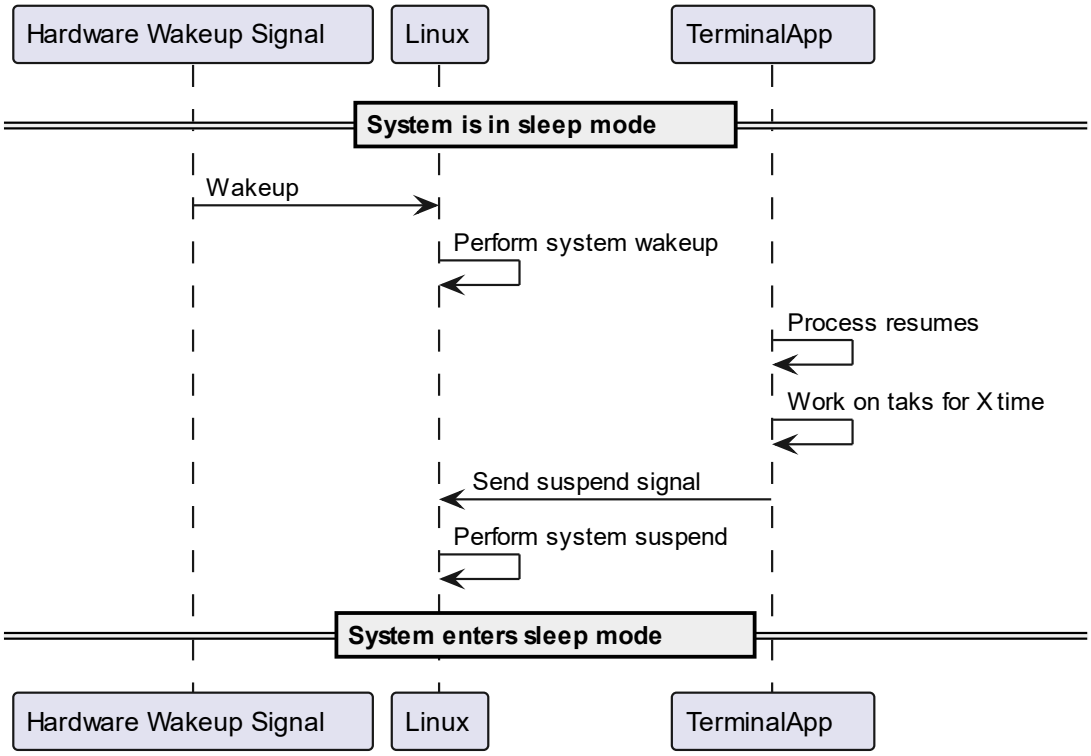

Figure 11. Wake up from a hardware signal.

Figure 11 illustrates the system's wake-up process. When a hardware signal wakes up the system, Linux will begin to re-initialize itself to the pre-suspend state. After which, TerminalApp resumes operation and performs its configured

tasks. The low-power module sets the system back to sleep when the minimum wake-up time has passed.

The digital input wake-up driver for WRM247LTE is responsible for configuring the WRM247LTE device's digital input pins wake-up capability. The driver supports configuring any device input pin to wake the system from a rising or falling edge interrupt. This was accomplished using the Linux driver's module parameter functionality that exposes the driver's internal values to the user space. TerminalApp software writes the desired configuration to the driver's module parameters. Then the driver configures the selected pins as wake-up interrupts when the system is going to sleep and deletes the interrupts when waking up. While it is suboptimal to delete and create the same interrupts during systems suspend and resume transitions, the module parameter method allowed for faster development and testing as the driver's operation could be modified from the device terminal. The operation of the driver is shown in figure 12.

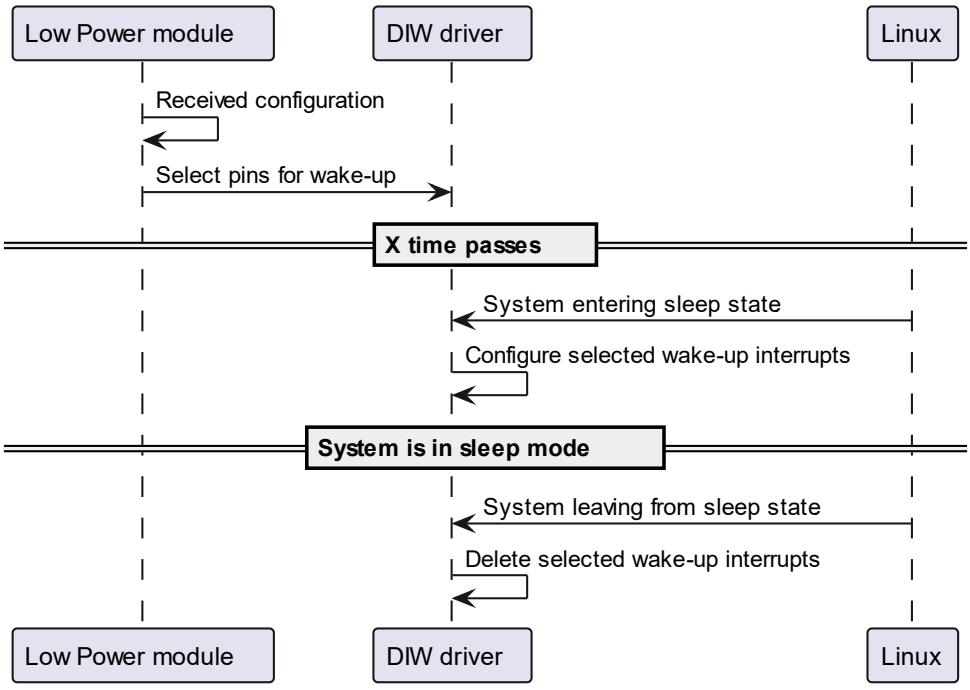

Figure 7. plantUML of Digital Input Wake-up driver operation.

As shown in figure 12, when the low-power module receives a new configuration, it sets the wake-up pins by modifying the driver’s module parameters. Based on the configuration, the driver then creates and deletes the system’s wake-up pins when the system enters sleep.

An alternative approach for configuring the wake-up interrupts would have been to use the Linux ioctl framework instead of module parameters. Pins would only need to be disabled and enabled during systems power management transitions, while the creation and deletion only when the wake-up configuration is changed. This approach would have been more challenging to develop and test, as sending ioctl commands from user-space to driver requires a separate program.

5 Conclusion

Initial planning for the WRM247LTE device's low-power features involved writing a few Linux drivers to handle the device's low-power groups and GPIO pin state in sleep. A more driver-oriented approach was taken to avoid controlling the hardware from separate parts of the system. This involved investigating how the various Linux drivers operate and other power management driver frameworks provided by the operating system.

To estimate the power consumption reductions, readings of the system's power consumption were taken using a multimeter's ammeter functionality. While the system's variable clock CPU and radio antennas made it challenging to evaluate minor runtime improvements, the multimeter readings were mainly used to measure the impact of power-usage improvements to the system's sleep state. Power usage was calculated by multiplying the power supply's voltage level and current draw.

Setting an idle system into suspend to ram state reduces the power usage from 720mW to 199mW, reducing the power usage by at least 72%. Waking the system up from the ram sleep state takes around 2 seconds. Restoring the system's network functionality takes longer depending on the networking interface, from 3 seconds on ethernet to 20 seconds on cellular. Overall power-saving results were good, and the start-up latency is acceptable. As the device begins collecting the data after wakeup, sending data to the cloud is not immediately essential.

Optimizations for the system's runtime power consumption ended up being minor. By default, the inactive hardware components are powered down. This was true for all hardware modules except the WLAN, which spent all its time in sleep mode. Keeping the WLAN powered down when not in use reduced the system's power consumption from the original reading of 732mW to 720mW, decreasing power usage by 1.6%.

While a large chunk of the work hours during the two-month timeslot allotted for the thesis was spent debugging hardware issues of ethernet transceivers, a working feature was created. Implementation of the sleep state produced good results, but there is still a tiny gap between the power usage results to the hardware engineers' specifications.

Future work will involve further reducing power usage and adding more ways to wake the system from sleep. Additional wakeup triggers can be implemented from messages on the CAN and RS-232 bus or readings from the acceleration sensor. In the future, Runtime power usage can also be improved by disabling the low-power groups during device use, enabling modems doze mode, or reducing radio signal strength.

References

1 ARM, Rob Aitken. Performance per Watt is the new Moore's law. [cited 3 November 2022] https://www.arm.com/blogs/blueprint/performance-per-watt

2 Embedded Computing, Omar Mohammed. There's high demand for low power. [cited 3 November 2022] https://embeddedcomputing.com/application/healthcare/personal-medical-devices/theres-high-demand-for-low-power

3 Design & Reuse. The evolving topology of SoC power management. [cited 1 November 2022] https://www.design-reuse.com/articles/9150/power-islands-the-evolving-topology-of-soc-power-management.html

4 Baylibre. An overview of generic power domains on Linux [cited 1 November 2022] https://baylibre.com/generic-power-domains

5 EmbeddedArtistry. Demystifying microcontroller GPIO settings. [cited 30 October 2022] https://embeddedartistry.com/blog/2018/06/04/demystifying-microcontroller-gpio-settings/

6 Silicon labs. What GPIO configuration should I use for unconnected pins to minimize current consumption? [cited 15 October 2022] https://community.silabs.com/s/article/what-configuration-should-i-use-for-unconnected-unused-gpio-pins-to-minimize-c?language=en_US

7 Texas Instruments. 1997. CMOS Power Consumption and Cpd calculation. https://www.ti.com/lit/an/scaa035b/scaa035b.pdf

8 AnandTech, Johan De Gelas. The quest for more processing power. [cited 1 November 2022] https://www.anandtech.com/show/1611/3

9 ARM. ARMv8-A Power management. [cited 15 October 2022] https://developer.arm.com/documentation/100960/0100/ARMv8-A-Power-management?lang=en

10 Luca Benini, Alessandro Bogliolo, Giovanni De Michelli, 2000. A survey of design techniques for system-level dynamic power management.

11 Intel. StrongARM® SA-1100 Microprocessor datasheet.

12 The Linux foundation. Admin guide. [cited 3 November 2022] https://elixir.bootlin.com/linux/v5.4.81/source/Documentation/admin-guide/README.rst

13 Redhat. Why understanding user space vs kernel space matters. [cited 1 November 2022] https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters

14 Kernel development community. Intro lecture. [Cited 1 November 2022] https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html

15 The Linux foundation. Kernel definition. [cited 3 November 2022] http://www.linfo.org/kernel.html

16 The Linux foundation. Device driver documentation. [Cited 1 November 2022] https://docs.kernel.org/driver-api/driver-model/driver.html

17 The Linux foundation. Device link documentation. [cited 3 November 2022] https://www.kernel.org/doc/html/v5.4/driver-api/device_link.html

18 The Linux foundation. Linux and the devicetree [Cited 1 November 2022] https://www.kernel.org/doc/html/latest/devicetree/usage-model.html

19 Octavo Systems. Linux device tree [Cited 1 November 2022] https://octavosystems.com/app_notes/osd335x-design-tutorial/osd335x-lesson-2-minimal-linux-boot/linux-device-tree/

20 The Linux foundation. PM driver types. [cited 3 November 2022] https://www.kernel.org/doc/html/v5.4/driver-api/pm/types.html

21 The Linux foundation. Freezing of tasks. [cited 3 November 2022] https://elixir.bootlin.com/linux/v5.4.81/source/Documentation/power/freezing-of-tasks.rst

22 Intel. How to achieve SOIX states on Linux. [Cited 1 November 2022] https://01.org/blogs/qwang59/2018/how-achieve-s0ix-states-linux

23 The Linux foundation. Pm states. [cited 3 November 2022] https://elixir.bootlin.com/linux/v5.4.81/source/Documentation/admin-guide/pm/sleep-states.rst