Energy Research and Development Division

FINAL PROJECT REPORT

Mobile Efficiency for Plug-Load Devices

Developing a Portfolio of Advanced Efficiency Solutions: Plug-Load Technologies and Approaches for Buildings - Phase II

California Energy Commission

Gavin Newsom, Governor

April 2019 | CEC-500-2019-044

PREPARED BY:

Primary Author(s):
Davorin Mista
Yefu Wang, Ph.D.
Joshua Kuhlmann
Vojin Zivojnovic, Ph.D.

AGGIOS, Inc
5251 California Ave
Irvine, CA 92673
Phone: 949-212-0130 | Fax: 949-212-0130
http://www.aggios.com

Contract Number: EPC-15-021

PREPARED FOR:
California Energy Commission

Felix Villanueva
Project Manager

Virginia Lew
Office Manager
ENERGY EFFICIENCY RESEARCH OFFICE

Laurie ten Hope
Deputy Director
ENERGY RESEARCH AND DEVELOPMENT DIVISION

Drew Bohan
Executive Director

DISCLAIMER

This report was prepared as the result of work sponsored by the California Energy Commission. It does not necessarily represent the views of the Energy Commission, its employees or the State of California. The Energy Commission, the State of California, its employees, contractors and subcontractors make no warranty, express or implied, and assume no legal liability for the information in this report; nor does any party represent that the uses of this information will not infringe upon privately owned rights. This report has not been approved or disapproved by the California Energy Commission nor has the California Energy Commission passed upon the accuracy or adequacy of the information in this report.

ACKNOWLEDGEMENTS

The authors acknowledge the vision, determination, and support of the California Energy Commission that initiated, motivated, guided, and funded the research in the efficiency of plug-load devices presented here.

PREFACE

The California Energy Commission's Energy Research and Development Division supports energy research and development programs to spur innovation in energy efficiency, renewable energy and advanced clean generation, energy-related environmental protection, energy transmission and distribution and transportation.

In 2012, the Electric Program Investment Charge (EPIC) was established by the California Public Utilities Commission to fund public investments in research to create and advance new energy solution, foster regional innovation, and bring ideas from the lab to the marketplace. The California Energy Commission and the state's three largest investor-owned utilities—Pacific Gas and Electric Company, San Diego Gas & Electric Company, and Southern California Edison Company—were selected to administer the EPIC funds and advance novel technologies, tools, and strategies that benefit their electric ratepayers.

The Energy Commission is committed to ensuring public participation in its research and development programs that promote greater reliability, lower costs, and increase safety for the California electric ratepayer and include:

- Providing societal benefits.
- Reducing greenhouse gas emission in the electricity sector at the lowest possible cost.
- Supporting California's loading order to meet energy needs first with energy efficiency and demand response, next with renewable energy (distributed generation and utility scale), and, finally, with clean, conventional electricity supply.
- Supporting low-emission vehicles and transportation.
- Providing economic development.
- Using ratepayer funds efficiently.

Mobile Efficiency for Plug-Load Devices is the final report of the project Developing a Portfolio of Advanced Efficiency Solutions: Plug-Load Technologies and Approaches for Buildings (Contract Number EPC-15-021) The information from this project contributes to Energy Research and Development Division's EPIC Program.

All figures and tables are the work of the author(s) for this project unless otherwise cited or credited.

For more information about the Energy Research and Development Division, please visit the Energy Commission's website at www.energy.ca.gov/research/ or contact the Energy Commission at 916-327-1551.

ABSTRACT

Why do mobile devices, like smart phones, use significantly less energy than plugged devices, like computers, while offering comparable functionality to users? This question naturally arises as users of mobile devices experience undistinguishable web browsing, video streaming, and other services from their battery-operated mobile devices. Can researchers learn from the design principles for mobile devices in order to close the mobile energy gap and save energy consumption for billions of plug-load devices?

Design principles define the technical steps engineers take to create new devices. The legacy design principle for plug-load devices is to focus on functionality and performance first and deal with energy efficiency later once the device is fully developed. This legacy principle results in vastly suboptimal energy efficiency of devices. Only the largest and most profitable mobile businesses, like Apple®, have been able to overcome this legacy and consistently follow energy-efficient design principles.

This project aims to save energy by applying the mobile design principles to plug loads and delivering four design principles for energy-efficient development of plug-load devices. These are the methodology document, technical standard, virtual prototype, and real-life reference design. With these principles, any engineers properly skilled in the design and production of standard electronic devices can assess the opportunities for energy savings.

Consistent use of the new technology can reduce the annual energy consumption of targeted home and business electronic plug-load devices in California's homes and offices by 20-50 percent and achieve statewide cumulative savings through 2022 of 1.6–4 terawatt-hours, depending on the range of addressable products and depth of per unit savings. (A terawatt is a trillion watts.)

Keywords: Energy efficiency, plug-load devices, miscellaneous electrical loads (MEL), unified hardware abstraction, software defined energy management

Please use the following citation for this report:

Author(s) Mista, Davorin, Yefu, Wang, Ph.D., Joshua, Kuhlman, Vojin, Zivojnovic, Ph.D. 2019. *Mobile efficiency for Plug-Load Devices*. California Energy Commission. Publication Number: CEC-500-2019-044.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS ..... **i**

PREFACE ..... **ii**

ABSTRACT ..... **iii**

TABLE OF CONTENTS ..... **iv**

LIST OF FIGURES ..... **vi**

LIST OF TABLES ..... **viii**

EXECUTIVE SUMMARY **1**

Introduction 1

Project Purpose 1

Project Process 2

Project Results 3

Benefits to California 4

CHAPTER 1: Overview **5**

CHAPTER 2: Methodology **6**

Energy-First Development Process 6

Power Modeling using UHA 6

UHA Language Elements 6

Modeling Hardware Components and Power States 7

Modeling Software and External Events 8

Modeling Dependencies 8

Modeling Latencies 9

Power Accounting in UHA 10

Optimizing Energy Efficiency on the Device 12

Power Measurements 12

Software Defined Energy Management 13

EEMI Interface 13

Overview 13

Use Cases 14

Optimized Power Management Firmware 15

CHAPTER 3: Virtual Prototypes **16**
High-Level Modeling 17
Set-Top-Box High-Level Modeling 17
PC High-Level Modeling 21
TV High-Level Modeling 23
Gaming Console High-Level Modeling 24
Subsystem Energy Modeling 24
Modeling Techniques 24
Power Models of Components 26
Latency Modeling 43
Latency Modeling Based on Datasheet 43
Latency Modeling from Logging Timestamps 43
User Behavior Modeling 44
Device Usage Patterns 44
Environment Impact Patterns 46
Simulation Results 47
TV Virtual Prototype 48
Computer Virtual Prototype 53
Set-Top-Box Virtual Prototype 56
Gaming Console Virtual Prototype 57
CHAPTER 4: Reference Designs **66**
Components Implementation 66
Hardware platform 66
Operating system and application software 67
Testbed 68
Hardware Tools 69
Software Tools for per-rail power measurements 70
Test Plan 71
Measurements and Power Saving Analysis 75
Latency Results 75
TV reference design power analysis 78
Computer reference design power analysis 79

Set-top-box reference design power analysis 82
Gaming console reference design power analysis 86
Summary of reference designs 88
CHAPTER 5: Conclusions 89
Further Work and Next Steps 89
CHAPTER 5: Technology Transfer 90
Standardization Efforts 90
Proliferation Efforts 90
REFERENCES 91
APPENDIX A: UHA Reference Manual A-1
APPENDIX B: UHA Modeling Guide B-1

LIST OF FIGURES

Page

Figure 1: UHA Objects and Properties 7
Figure 2: Power Accounting in UHA Models 11
Figure 3: UML Diagram for Shared Memory Use Case 14
Figure 4: A Table Listing Typical Power Consumption Values From a USB Controller IC (Microchip, 2017) 25
Figure 5: Power Model of N25Q128A 25
Figure 6: USB Host Setup 29
Figure 7: Measurement Setup for Building the Power Model of the LED Backlight 33
Figure 8: Back Light Power Model 33
Figure 9: TV Power Drops After a Sudden Move to a Completely Dark Environment 34
Figure 10: Back Light Power Consumption in Dark or Bright Rooms 35
Figure 11: Power Model of the Amplifier 36
Figure 12: Gaming Console Usage Distribution Used in Virtual Prototyping 45
Figure 13: Use Distribution in Set-Top Box Virtual Prototype 46
Figure 14: Modeling Ambient Light Levels 47

Figure 15: Power Consumption of the Virtual Prototype Under the Traditional TV Use Case 48

Figure 16: Ambient Light Level in the Traditional TV Use Case 49

Figure 17: Power Breakdown in the Traditional TV Use Case 49

Figure 18: Smart TV Use Case 50

Figure 19: YouTube App in the Virtual Prototype Compared With the Physical Measurement 52

Figure 20: Power Consumed in the USB Gallery Use Case 53

Figure 21: Power and Latency Graph for a 24-Hour Cycle of the Ideal ENERGY STAR®-PC Model 54

Figure 22: Power and Latency Graph for a 24-Hour Cycle of the Zynq MPSoC-Based Model 55

Figure 23: An Enlargement of an Activity Cycle Based on Sporadic User Activity 56

Figure 24: Feature Demos of the Set-Top Box Virtual Prototype 57

Figure 25: Power Consumption Comparison Between the Gaming Console Virtual Prototype and a Typical Gaming Console 57

Figure 26: Per Component Power Savings of the Gaming Console Virtual Prototype in Standby Mode 59

Figure 27: Per Component Power Savings of the Gaming Console Virtual Prototype in Navigation Mode 59

Figure 28: Per Component Power Savings of the Gaming Console Virtual Prototype in Streaming Mode 60

Figure 29: Per Component Power Savings of the Gaming Console Virtual Prototype in Gaming Mode 60

Figure 30: Power and Latency Graph for a Streaming-Heavy 96-Hour Cycle of the Reference Gaming Console Model 61

Figure 31: Power and Latency Graph for a Typical 24-Hour Cycle of the Reference Gaming Console Model 62

Figure 32: Power and Latency Graph for a Streaming-Heavy 96-Hour Cycle of the Zynq MPSoC-Based Model 62

Figure 33: Yearly Power Usage of the Gaming Console Virtual Prototype 63

Figure 34: Power and Latency Graph for a Normal 24-Hour Cycle of the Zynq MPSoC-Based Model 63

Figure 35: Architecture Diagram of the Set-Top Box Reference Design 67

Figure 36: Reference Design Testing Set-up 70

Figure 37: Measurement Diagram of ZCU102 Rail Measurement System 71

Figure 39: Hardware Latency Breakdown by Component: Full-Power Domain, DDR SDRAM, and Application Processing Unit 77

Figure 40: Software Latency Breakdown Based on Logged Linux information 77

Figure 41: Total Latency Breakdown Into Hardware and Software Latency 77

Figure 42: USB Gallery: Power Consumption With Standard Deviation 79

Figure 43: Power Consumption of the Reference Design Compared With a Traditional PC ... 81

Figure 44: Assumed Usage Pattern in 14 Hours 82

Figure 45: Power Comparison Average to a Year 82

Figure 46: Set-Top Box Reference Design Power Measurements 83

Figure 47: DC Power Measurement of ZCU100 84

Figure 48: Comparison of Reference Design and the Baseline 86

Figure 49: Comparison of Reference Design to Traditional Console 87

LIST OF TABLES

Page

Table 1: Resulting System Power Consumption 12

Table 2: States in TV High-Level Modeling 23

Table 3: Power States of the Flash Chip 27

Table 4: ULPI Power States 29

Table 5: Wi-Fi Receiver States 30

Table 6: Wi-Fi Transmitter States 32

Table 7: States of the Amplifier Driving the Speaker 37

Table 8: Power Models of the HDMI Host 38

Table 9: Comparisons With the Samsung TV, Excluding the Screen and the Back Light 51

Table 10: Expected Power Savings From the Gaming Console Virtual Prototype 58

Table 11: Power Savings of the Gaming Console Virtual Prototype 64

Table 12: Power Consumption and Savings of Computer Reference Design 80

Table 13: Breakdown of Power Consumption by Component 84
Table 14: Set-Top Box Power and Energy Savings 85
Table 15: Gaming ConsolePower and Energy Savings 87

EXECUTIVE SUMMARY

Introduction

Market leaders in mobile products, like California-based Apple®, have developed battery-powered electronic devices with highly sophisticated functionality and minimal energy consumption. The intriguing question is how the best design practices from the mobile industry leaders, like Apple, can be applied to improve the energy efficiency of plug-load devices like TVs, computers, game consoles and set-top boxes. Lower energy consumption of plug loads leads to increased building energy efficiency and is essential to reach California's ambitious energy efficiency goals.

Energy management of plug-load devices is either provided as part of the proprietary hardware and software integrated in the device by in-house power design teams or is controlled externally through independently supplied add-on devices that cut off the power supply (e.g. advanced power strips). Whereas the former approach comes with high costs for specialized components, software, and design know-how, the latter is prohibitive due to long recovery times for the device after the power is restored, which results in long wake-up latencies not tolerated by the users.

Apple and other mobile market leaders have achieved extraordinary results in mobile energy efficiency and even ported their mobile efficiency principles to wall-powered device design. However, the vast majority of other plug-load devices are designed by outsourcing companies that are not skilled in energy efficiency technologies. This project contributes to the closure of the mobile energy gap by developing design principles, technical standards, virtual prototypes, and reference designs that will enable most plug-load development teams to achieve best-in-class mobile efficiency levels within short time and at low costs.

Project Purpose

This project seeks to reduce the energy consumption of plug-load devices in California's homes and offices by applying mobile design practices. These practices include the adoption of the energy proportionality principle, selection of energy- and cost-efficient hardware components, and implementation of specialized energy management software kernels that monitor and control device power. Based on these innovations, the objective is to develop, tune, and deploy best practice guidelines in form of methodology documents and reference designs for energy-efficient plug-load designs to the manufacturers of plug-load devices and their hardware suppliers, software suppliers, and regulators.

The centerpiece of the project is the development, formal standardization, and industrywide deployment of the technical standard for energy proportional computing called *Unified Hardware Abstraction and Layer for Energy Proportional Electronic Systems*. The proposal of the new standard is submitted to the IEEE P2415 Working Group and is supposed to be adopted by IEEE by the end of 2020. IEEE P2415 defines the syntax and semantics for energy-oriented description of hardware, software and power management for electronic systems. It enables

specifying, modeling, verifying, designing, managing, testing, and measuring the energy features of the device, covering the pre- and postsilicon design flow.

Project Process

The main approach to achieving energy savings in plug-load devices is to follow the *energy-first design principle*. The longer development engineers postpone tuning the power aspects of the devices, the more difficult it becomes to alleviate inefficiencies, resulting either in increased cost or in less optimal energy efficiency. So, rather than implementing functionality first and tuning power once the functionality is complete, the project approach is to take energy into account at every step of development.

The first step is to do energy modeling using virtual prototypes. An energy-oriented virtual prototype allows simulation of the power states of the complete device for a set of use cases. This kind of simulation provides early insights into the expected energy profile that a device might have for different types of use models. It also allows enhancing optimizing design aspects, and software partitioning which decides how the components of a system are implemented in hardware and which ones in software for energy efficiency.

Once the actual hardware platform becomes available, it is important to continue following the energy-first development method:

- Power measurements should be conducted at all times during all steps of development.
- Software defined energy management principles should be followed.
    - Power control should be separated from data processing.
    - Power control and monitoring should be delegated to a dedicated entity in the system.
    - Power control interfaces should be abstracted to allow different software layers to issue power-related requests.

Following these principles, the research team created reference designs for each of the four device types. The reference designs were based on the Xilinx ZCU102 prototyping platform using standard software components that have been extended to include an energy management interface that satisfies the software-defined energy management principles outlined above.

Optimizing energy use in a real device can be daunting, given the complexity of today's systems, both on the hardware side with hundreds of interdependent components, as well as on the software side, with layered software architectures and multiple independent processors running different software applications often developed by different teams. Small changes in the software can often have big impacts on the power consumption of a device.

A crucial factor affecting the ability to optimize the energy behavior of a device is understanding where power is consumed at any given moment. Accurate and fine-grained power measurements are key for gaining this understanding. On the reference designs

developed for this project, the research team measured power using measurement circuitry that is embedded on the development board used, providing detailed visibility into the power consumption of individual voltage rails.

Using the detailed visibility of power consumption on the real device with the knowledge gained on the virtual platforms, opportunities for energy efficiency optimizations were easily seized. This ensured that in any given application state, each hardware component of the system was in the lowest possible power state, satisfying functional and latency requirements of the respective application.

Throughout this project, the design method has been refined and extended, while the unified hardware description, which is a standardized language, was used to model electronic systems. This format was tested by applying it to the different designs and at different levels of abstraction which is the process of removing physical, and spatial details to study the attributes of an electronic system. A key objective was to produce a description format for energy models that meets the needs of software and system developers, including expressive power and readability, while allowing reuse of models as well as collaboration between teams. Reuse is especially important since a system designer will want to source the energy models from their IP and chip vendors rather than describing all components from the ground up themselves.

Ultimately, the work done during this project helped produce the current version of the unified hardware description, which has been submitted to the IEEE P2415 working group for standardization.

Project Results

Using the proposed method, the team created virtual prototypes of each device and has implemented reference designs based on a commercially available development board. For each device that has been targeted, the final reference designs demonstrated significant energy savings compared to the typical devices.

Effective energy modeling and virtual prototyping require a suitable description format to capture the energy-related behavior of a system and complete device. For this purpose, the unified hardware abstraction layer has been refined and extended based on the experience gained throughout this project, culminating in the donation of the layer to the IEEE P2415 working group. The IEEE standardization efforts are a key factor in enabling wider adoption of the design method proposed in this project. The design method provides industrywide agreement on a description format, promoting information exchange and reuse of models across teams and between suppliers and system integrators. The electronic design tools produced by industry increase the versatility of the energy models, similar to how the language standards from the 1980s standardized how functional aspects of hardware were described, giving birth to a multibillion-dollar design automation industry.

Technology Transfer

One part of the technology transfer effort was to standardized the unified hardware abstraction through the IEEE standardization body. The status of the standardization is that a final proposal has been submitted to the P2415 working group. To promote the developed technology in front of the industry, the findings from this project have been presented at several forums.On February 8, 2018, AGGIOS was invited by NRDC to present the findings from the Electric Program Investment Charge (EPIC) mobile efficiency project to the set-top box community during their STB voluntary agreement meeting in Denver. On August 15, 2018, AGGIOS held an emerging technologies webinar hosted by the California Energy Commission to present the results of the EPIC project to a wide audience from the industry as well as regulators. The webinar had more than 140 attendees.

Benefits to California

Considering the large numbers of TVs, set-top boxes, computers, gaming consoles, and other plug-load devices, the energy use in California can be reduced significantly if the energy-first design method developed in this project is adopted by industry.

This research does not provide increased ratepayer reliability or safety; however, it can lower customer electric bills, and can add to a significant contribution to California's statutory greenhouse gas reductions and energy efficiency goals. At a large scale, this project holds the potential for long-term, deep savings across a broad array of plug-load devices.

Consistent deployment of the new technology has the potential to reduce the annual energy consumption of targeted electronic plug-load devices in California homes and offices by 20-50 percent and achieve statewide cumulative savings through 2022 of 1.6–4 terawatt-hours (TWh), depending on the range of addressable products and depth of per-unit savings. (A terawatt is one trillion watts.)

This project will additionally accelerate the use of mobile efficiency technologies across product categories by influencing policy mechanisms to accelerate adoption of mobile efficiency technologies. These policy pathways include:

California Title 20: The results and implementation of this work can enable the Energy Commission to revise energy allowance values downward resulting in significant savings relative to today's proposed levels.

Voluntary Agreements: Results from this project can be used to renegotiate voluntary agreements or influence the stringency of future tiers of electronic standards.

ENERGY STAR®: As mobile efficient wall-powered devices begin to appear in the market, ENERGY STAR can set appropriate allowance levels to accelerate the adoption of mobile efficiency technology in wall-powered devices. ENERGY STAR can also list the future Unified Hardware Abstraction IEEE P2415 electronic standard compliance as a requirement to support this industrywide standards effort.

Utility Incentive Programs: Utility program s can use incentives to accelerate the adoption of ENERGY STAR products based on mobile efficiency.

CHAPTER 1: Overview

Miscellaneous plug loads constitute a significant portion of the energy consumption in California's households. The capabilities of many of these devices are converging as they often are built on top of similar platforms. This project focuses on the energy efficiency of personal computers, gaming consoles, TVs as well as set-top-boxes. While each of these devices has a unique functionality, the devices have a large set of common capabilities. Those same capabilities are also offered by many mobile devices, which often have similar processing power but consume far less energy.

This project explores and quantifies how an improved development method can help improve the energy efficiency of these plug-load devices.

CHAPTER 2: Method

When it comes to mobile devices, the industry has already understood that energy efficiency is most effectively achieved when including it as a key criterion early in the design process, producing remarkable results in terms of energy efficiency without sacrificing performance. Cost is often cited as a reason why traditional plug-load devices ignore energy efficiency. This project aims to demonstrate that when using the right method, energy efficiency can be achieved without significantly increasing development costs.

This chapter describes the method used for developing energy-efficient devices throughout this project. It consists of an energy-first development process starting with virtual prototypes, followed by using software-defined energy management principles on the final device alongside detailed power measurements throughout development.

Energy-First Development

Most plug-load devices are being developed with a focus on functionality and performance. Energy efficiency either is often neglected entirely or is introduced in the late stage of product development. Consequently, power savings are often difficult to achieve without requiring a redesign of parts of the hardware and software that has already been finalized, resulting in substandard energy efficiency.

The energy-first development process that has been applied throughout this project starts with modeling and simulation, followed by constant measurement and monitoring of power consumption during the implementation phase.

Power Modeling Using Unified Hardware Abstraction

To create suitable models of the energy behavior of the devices in question, the research team used the Unified Hardware Abstraction (UHA) language. UHA descriptions consist of

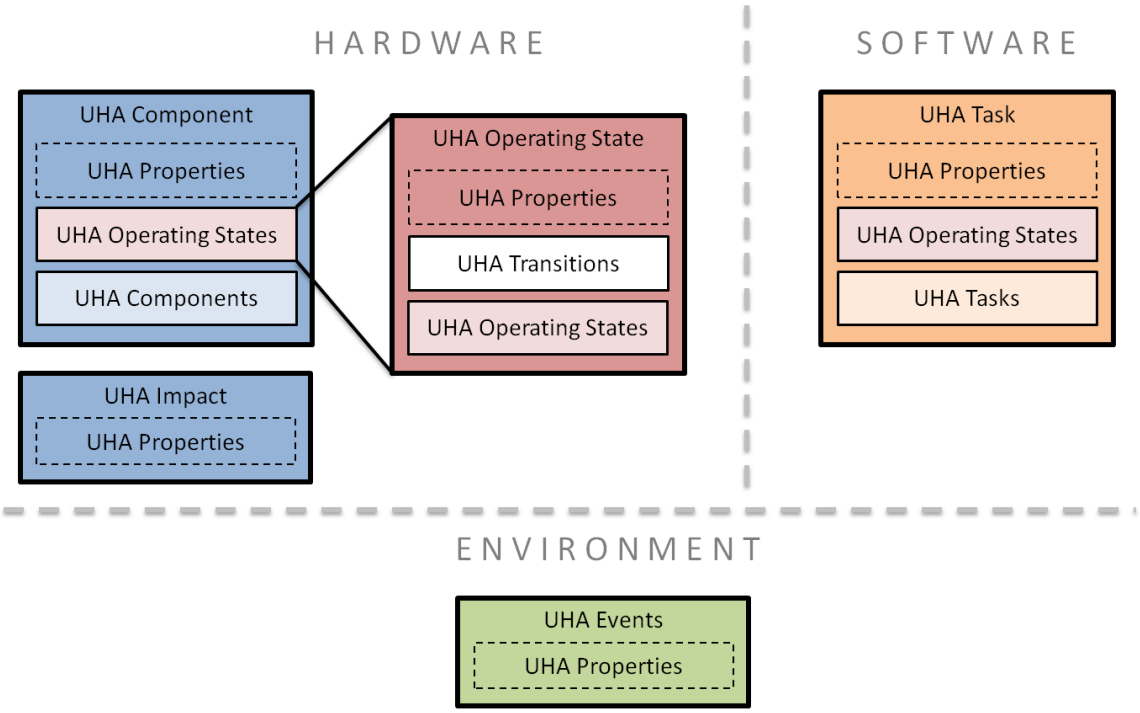
- Models of hardware components and related power states.
- Models of software and external events.
- Dependencies between the software and components.

By modeling the power states of components, as well as the dependencies between the components and the dependencies between software and the hardware, the power consumption of the overall system can be accurately estimated.

UHA Language Elements

The main language elements of UHA are objects and properties. UHA objects describe objects such as components, software tasks, or power states such as on and off. UHA properties describe attributes of an object.

Figure 1: UHA Objects and Properties



Source: Aggios, Inc.

The syntax of UHA is based on the device tree syntax. references (Linaro Limited, 2016): (https://elinux.org/Device_Tree_Reference ; https://www.nxp.com/docs/en/application-note/AN5125.pdf).

Beyond the device tree syntax, UHA has support for annotating object types and run-time parameters, as well as support for arithmetic expressions and lookup tables.

Modeling Hardware Components and Power States

Hardware components are modeled as UHA objects. Hardware components can be hierarchical; in other words, a hardware component can have subcomponents. A special type of UHA object is the power state, or UHA state. Each component can have one or more states. States themselves can also be hierarchical.

Each component can have one or more power models as a UHA property. Power models can be specific to a state. Power models can reference properties defined anywhere in the model, that is, component-specific properties or systemwide properties.

UHA transitions are used to describe the behavior of a component when a new state is entered. This includes the related energy profile, as well as the transition latency, i.e. the time it takes to perform the transition from the previous state to the new state.

UHA also allows modeling of “control components,” i.e. clocks, voltage rails, or reset lines. Control components are an important detail for modeling the power states of digital systems as most power states are controlled by either clocks, voltage rails, or reset lines. Control components typically form hierarchies of their own, such as the clock tree or the voltage hierarchy.

Modeling Software and External Events

In UHA, software is modeled as UHA tasks, while the influence of hardware events is modeled as UHA impacts. UHA tasks and impacts may be hierarchical and they may have states. (E.g., an operating system task is active or in a suspended state.) Software states are important in determining overall transition latencies as those latencies may be the gating factor for being able to enter low power states during periods of inactivity.

Modeling Dependencies

Dependencies are a crucial aspect of UHA models, ensuring that only valid combinations of states are being considered when simulating the energy-related behavior of a system.

UHA dependencies for software tasks describe the needs of a software task with respect to states of hardware components. UHA dependencies also extend to relationships between hardware components, specifically between the states of components.

Explicit Dependencies

Dependencies can be expressed explicitly using the “requires” property.

```
cpu {
   state on {
         requires = &bus/active;
   }
}
```

The above example shows a simple explicit dependency between two hardware components. It specifies that for the “cpu” component to be in the “on” state, the “bus” component must be in the “active” state.

More complex dependencies can be expressed as well, such as the requirement on a specific set of capabilities, or numeric requirements, for example, on a quality of service capability or a certain memory throughput capacity. Dependencies on latencies can be expressed as well.

Control Dependencies

Control dependencies are established by identifying the respective control parent(s) of a component and then describing the required state of the control parents for a given state of the component. Control dependencies are especially important when one or more components share the same clock or the same voltage rail, effectively linking the power states of several components despite those components potentially being functionally independent.

Control dependencies can also refer to clock frequencies or voltage levels rather than just states.

Modeling Latencies

Modeling latencies in UHA serve two purposes. On one hand, latencies capture the delays incurred when state transitions are performed, which allows tools to report and analyze such latencies. On the other hand, UHA also allows expressing dependencies in terms of latency requirements.

Latencies are modeled in UHA by defining state transition latencies for each component in the hierarchy. The transition latencies captured for one component shall account only for the time it takes for the component to transition from the previous state to the new state—they should not include the latency of other state changes triggered by any of the components requirements, such as changing of a clock frequency or turning on a clock or voltage source.

The total latency for the state transition of a components can be computed by aggregating the latency of the state transition with the latencies of all state transitions resulting from the dependencies.

Defining Entry and Exit Latency Properties

A UHA state may define entry and exit latency properties. The *transition latency* is computed by adding the exit latency of the old state or source state and the entry latency of the new state or target state.

```
eth0 {
   state on {
         entry-latency = 8ms;
         }
         state idle { }
         state off {
               exit-latency = 5ms;
         }
}
```

The code above describes a component with the respective transition latencies. The total latency for entering the on state is either 8 millisecond (ms) when transitioning from the idle state or 13 ms when transitioning from the off state.

Defining Latencies Within a Transition

When transitions are defined explicitly, they may also contain a latency property. This property explicitly specifies the transition latency between pairs of states, i.e., the source and target state.

```
memory {
   state on {
         transition {      // transition from any state
               latency = 3ms;
         }
         transition {      // transition from state "off"
               from = &off;
               latency = 12ms;
}  }     }
```

In the example above, because the transition from source state “off” is defined explicitly, it supersedes the generic transition specified without a “from” property. Hence, the transition latency into the “on” state is either 12 ms from the off state or 3 ms for all other cases.

Power Accounting in UHA

Component power information in UHA is defined on multiple levels. To compute the total power consumption for components and systems, a simple three-step process should be followed.

- The first step is to select a power model for each component.
- The second step is to compute the power for each component.
- The third step is to aggregate the total power of the system or subsystem of interest.

The subsequent sections explain each of the steps in more detail.

Power Model Selection

Each component shall have only one power model at any instance in time. The rules for selecting the active power model based on the current state are as follows:

Power models specified on the component level are superseded by power models specified in the current state of the component. If no state-specific power model exists for the current state, then the power model defined at the component level is the one that applies.

If there is a hierarchy of states defined, the power model of the leaf state always trumps the power model of the parent(s) state. During a transition, a power model specified in the transition object trumps all other power models for that component.

Power Value Calculation

For each component, the respective power value is determined using the selected power model and the current values of the referenced properties.

Power Value Aggregation

The total power is aggregated hierarchically. Hence, for a given component or (sub)system, the total power is computed by adding the power of all children to own power value of the component. Finally, the power values of the active tasks and impacts related to this (sub)system are also added.

For example, to account for the total power consumption of the system, the power values for all the components and related sub-components, the power values of the active segment of the energy profile of the running tasks, and the power values of the active impacts need to be added.

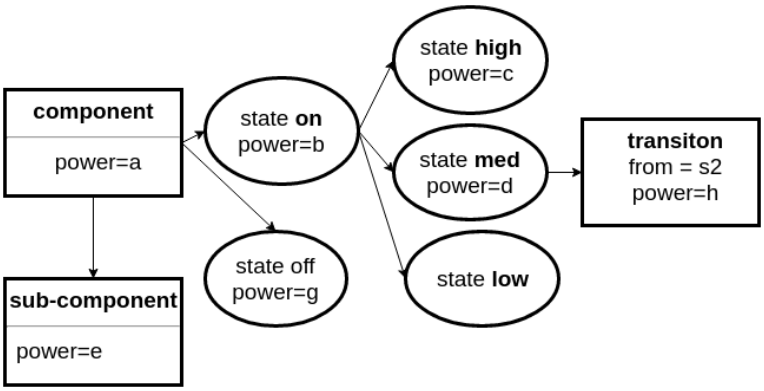
Examples

Table 1: Resulting System Power Consumption

shows an example of a simple model with hierarchical power states and multiple power models. The power accounting and the resulting power consumption for the complete system in different states are in **Error! Reference source not found.**.

The power model for "component" is selected based on the related state, or the transition if the state is in the transition and the transition has the power property defined. Then the total power consumption needs to be added to the power consumption of the associated subcomponent.

Figure 2: Power Accounting in UHA Models



Source: Aggios, Inc.

Table 1: Resulting System Power Consumption

Component state	*Total power consumption*
High	c+e
Med	d+e
Low	b+e
Off	g+e
Transition from “off” to “med”	h+e
Transition from “off” to “high”	c+e

Source: Aggios, Inc.

Optimizing Energy Efficiency on the Device

Once the development shifts to the real device, the method used throughout this project consists of three factors

- Power measurements at every step
- Optimized power management control firmware
- Software defined energy management throughout the software stack

Each of the three factors is equally important when looking to achieve optimized energy efficiency.

Power Measurements

Probably the easiest and most effective way to ensure energy efficiency is achieved through awareness of the energy profile of a device at all stages of development. Ideally one would have access to highly accurate and granular power data at all times. However, even high-level power measurements using entry-level equipment (e.g. alternating current [AC] power of the entire system) will allow engineers to spot unexpected changes in the energy profile and take corrective action early. Efficiency problems are best fixed at the moment when they’re introduced, as it is likely that the responsible software component can be easily identified.

If many software changes are introduced across the system at once, it can be significantly more difficult to isolate the software component that introduced the problem.

Having the ability to see real-time power data for a device is even more powerful when the power data are broken down by component. In most cases, this is impossible because many components are embedded in a single system on a chip (SoC) and share common voltage rails, which are supplied by the board. Still, when modern SoCs have several independent voltage rails, it is helpful when power measurements are done on a per-rail basis. In the case of the Xilinx ZCU102 board used for this project, there are 18 voltage rails that deliver power to the Xilinx UltraScale+ MPSoC, which is the main SoC of the system. Being able to measure the power consumption of each of these rails makes it easier to associate changes in power consumption to specific components, or at least to small sets of components. This can significantly reduce

the effort required to identify the components that may be to blame for changes in power consumption. This reduction in effort, in turn, reduces the amount of time and effort needed to analyze the power states of those components and ensure that each is in the lowest possible power state given the requirements of the application.

Software Defined Energy Management

Complex systems used in modern plug loads, such as the devices investigated in this project, usually include more than one operating system, running on more than one heterogeneous set of processors. While the most complex tasks of a given application usually require the most powerful processors to be involved and the main operating system such as Linux to be running, many routine and maintenance tasks can be delegated to smaller processors. However, when multiple software components are running independently of each other and aren't running on top of a single monolithic operating system, it is no longer possible to rely on the big operating system to be in charge of all power management decisions.

Software defined power management is based on the following principles:

- Separation of power control from data processing
- Delegation of power control and monitoring to a dedicated entity in the system
- Abstraction of power control interfaces to allow software to communicate the power state requirements

Applying software defined power management principles to today's plug-load devices requires interfaces across the different layers of the software, including standard operating systems such as Linux.

EEMI Interface

To allow coordination of power management requests across multiple processing units and operating systems an interface is required that follows the software defined power management principles. While standard interfaces for power management exist, they are mostly still targeting a single operating system (United EFI Forum, 2017) (ARM Limited, 2017).

In collaboration with Xilinx, the embedded energy management interface (EEMI) (Xilinx, 2017), has been developed to support the needs of heterogeneous multicore platforms such as the Xilinx UltraScale+ MPSoC (Xilinx, 2017).

Overview

The EEMI allows software components running across different processing clusters on a chip or device to communicate with a power management controller (PMC) on a device to issue or respond to power management requests.

The EEMI APIs allow one or more processing clusters to manage respective power states, systemwide power states, as well as the power states of slave devices, such as peripherals or memories. EEMI requests are submitted by processing clusters and serviced by the power management controller.

Use Cases

The following basic use cases illustrate the purpose and necessity of the EEMI APIs.

Use Case 1: Suspending Linux

When Linux is suspended, the processor it is running on will enter a sleep state. In more complex systems, entering the sleep state of a processor often requires a complex power-down sequence that cannot be safely performed by the affected processor itself.

EEMI offers a “self-suspend” API that tells the power management controller to perform the appropriate sleep-state handling for the affected processing cluster.

Use Case 2: Shared Memory

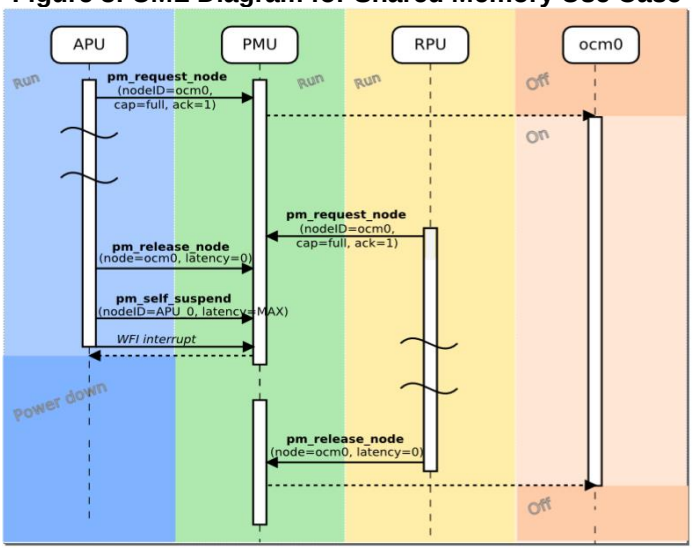
When two processing clusters are using the same memory, then neither entity would be able to safe place that memory into a low-power mode as it isn’t aware of the requirements of the other processing cluster with respect to that memory. EEMI includes API functions that allow processing clusters to declare when they need a certain resource and which capabilities they require.

Error! Reference source not found. shows the UML diagram including the EEMI API calls for this use case on the Xilinx UltraScale+ MPSoC.

Acronyms used in the UML diagram:

- APU: Application processing unit (Quad-core ARM Cortex A53)
- RPU: Real-time processing unit (Dual-core ARM Cortex R5)
- PMU: Power management unit
- OCM0: On-chip memory bank 0, i.e. the memory being shared

Figure 3: UML Diagram for Shared Memory Use Case



Source: Aggios, Inc.

A brief step-by-step description of the use case is outlined below:

Step 1: APU requests OCM0 by calling the pm_request_node API, which prompts the power management unit to power on the OCM0 memory bank.

Step 2: RPU also requests OCM0. Since the memory bank is already powered on, no action is required by the PMU.

Step 3: APU releases OCM0 by calling pm_release_node. This lets the PMU know that the APU no longer requires OCM0. However, no state change takes place because the PMU knows that the RPU still requires OCM0 to be on.

Step 4: RPU also releases OCM0. Because at this point the RPU was the last remaining user of the memory bank, the PMU will now turn off OCM0.

Use Case 3: System Shutdown

System shutdown is another use case that is well understood for simple architectures running a single operating system. However, on a heterogeneous platform with multiple independent software applications, a graceful system shutdown requires a certain degree of coordination. EEMI includes interfaces that enable adequately privileged entities to request a shutdown of a subsystem or the entire system or device. These interfaces inform all processing clusters about the requested system shutdown, allowing them to take appropriate measures to gracefully perform the respective shutdown sequences.

Optimized Power Management Firmware

After improving the power behavior of the system using the virtual prototype, UHA power models can also be used to customize the power management firmware for the target. The information contained in a UHA description can include all details necessary to generate the complete power management firmware. However, for the Xilinx UltraScale+ MPSoC that was used for the reference designs, a handwritten firmware already existed so that the UHA description was used to configure the PMU firmware and enable specific optimizations.

Possible optimizations range from lower power states for certain peripherals when those are unused to extended frequency scaling. Applying these optimizations operating systems like Linux typically perform on the main CPUs by lowering the bus frequencies as well as the DDR frequency during periods of low CPU frequency. Due to the already low energy consumption of ARM CPUs, pure CPU frequency scaling shows little noticeable benefit when no load is applied, whereas scaling of bus and memory frequencies can increase the savings by more than an order of magnitude.

CHAPTER 3:
Virtual Prototypes

Today, many product development teams for plug-load devices concentrate on functionalities first, measure the power consumption of the system only after they have a first physical implementation, and then try to enhance power consumption if the measured power is undesirable. At this stage in development, it is often impractical to make significant changes to the hardware or software design due to engineering difficulties or product timeline constraints. Therefore, many products are released with barely minimum power optimization to cover the heat dissipation needs.

The use of virtual prototyping is one of the critical design principles advocated that addresses this problem. Virtual prototyping is a design method that models the target system using power modeling tools at a very early stage, typically before the hardware is designed. The modeling involves the hardware and the software, targeting at typical usage scenarios.

The virtual prototype can be organized as two stages. In the first stage, in the high-level modeling stage, device models are built by modeling the high-level components. The power models are derived from the measurements of similar devices, from estimates based on the engineer's experiences or from regulatory requirements. The high-level modeling gives an estimated bound of the power consumption and provides reasonable expectations on the power consumptions of the final product. In the second stage, detailed virtual prototyping shall be conducted with the information on:

1. Power models on specific components. The high-level components should be split down to subcomponents with the detail that allows accurate modeling of the power, latency, and dependency with other components.
2. Latency modeling on specific components. Most plug-load devices will benefit from power optimizing algorithms with latency considerations. For example, if all components of a device have a very short latency when switched from the off state to the active state, the device has the potential to be put in the off state in short periods when the user is not using the device.
3. User behavior modeling. The way the user uses the device has enormous effects on the power consumption of the device.
4. Dependency modeling. In many cases, merely modeling the components of the device is not sufficient to derive an accurate power and latency model of the system. Many components have dependencies for it to operate correctly. For example, a hard drive on a gaming console has dependencies on the SATA controller in the system, either on the motherboard or as part of the system-on-chip (SoC). The SATA controller, in turn, has dependencies on the respective power network and clock network. If one were to

consider only the power consumption of the hard drive itself without accounting for these dependencies, it would be an underestimate.

The high-level modeling and the detailed modeling are optional. For some devices, the high-level modeling is sufficient—for example, when the device is modified from an existing device with a small number of changes, or the lack of information makes the detailed modeling too expensive. For other devices, the high-level modeling is unnecessary when there is already enough information to conduct detailed modeling.

The simulations of all the virtual prototypes in this chapter is performed with EnergyLab Version 0.2.001.

High-Level Modeling

The research team conducted high-level modeling on the targeted plug-load devices for several purposes. The first goal was to demonstrate how ENERGY STAR®-compliant devices operate under the standard test procedures defined in the ENERGY STAR standard. Second, the devices were split into high-level subsystems to pinpoint those subsystems that consume a significant amount of power or the subsystem with the most promising power savings potential.

Set-Top Box High-Level Modeling

The high-level model of the set-top box captures the ENERGY STAR®-typical energy consumption (TEC) budgets and adders (EnergyStar, 2014), along with the TEC power states and design choices for the various features that a set-top-box device may include.

The model has to capture:

- The power states of the device.
    - TV_watch, i.e. the active state.
    - Sleep, i.e. the basic inactive state.
    - APD, i.e. the auto-power-down state.
    - Sched_sleep, i.e. the scheduled sleep state.
- The TEC base allowance corresponding to the different types of devices covered by the ENERGY STAR standard.
- The TEC adders corresponding to the different features a set-top box device may have.
- The assumed daily distribution of device states making up the TEC allowance.

Modeling Power States

Each power state defined in the ENERGY STAR®-specification is represented by a UHA state.

```
state active {
    power = TBD;
```

```
        transition {
            from = &sleep;
            latency = 15s;
        }
        transition {
            from = &deepsleep;
            latency = 15s;
        }
    }
```

As the power levels are not prescribed by the ENERGY STAR®-specification, the power levels are left undefined using the “TBD (To Be Determined)” construct.

The transition latencies from the other power states are specified explicitly inside UHA transitions.

Even without explicitly defined power levels, it is possible to express relationships between power levels of different states using assertions.

Maximum allowed power levels for a particular state for example are described as in the following example:

```
    state sched_sleep {
        power = TBD;
        maxpower = &active/power_max*0.15;
        assert = <power LE maxpower>;
        assert = <power LE 3W>;
    }
```

Furthermore, for each state a corresponding task has been defined that represents the requirements for a particular device state.

Modeling TEC Power Allowances

The TEC base allowance is described by defining the different types of devices and then asserting that only one of them may be instantiated:

```
device {
    requires = &cable/present XOR
               &sat/present XOR
               &mvpd/present XOR …

    cable {
        state present { power = 50kWh / 365 / 24h; }
        state ignore { power = 0; }
    }
```

Using the “requires” property as well as the XOR operator, one can assert that exactly one of the listed devices must be present.

Each device has an associated TEC base allowance defined using the power model in the “present” state.

The TEC adders defined in the ENERGY STAR®-specification are described as UHA components. As ENERGY STAR®-does not prescribe power levels by power state for those additional features, the entire the TEC adder figure is converted to a power number for the component.

Requirements and assertions are used to express the dependencies or mutual exclusions for sets of features, such as in the case of the DOCSIS (CableLabs, 2017) support:

```
    docsis {
        requires = &docsis0/present XOR
            &docsis2/present XOR
            &docsis3/present;

        docsis0 {
            state present { power = 0; }
        }
        docsis2 {
            state present { power = 25kW / 365 / 24; }
```

```
            state ignore { power = 0; }
        }
        docsis3 {
            state present { power = 45kW / 365 / 24; }
            state ignore { power = 0; }
        }
    }
```

Modeling the TEC State Profiles

The ENERGY STAR®-specification defines specific state profiles that are to be used to determine the energy consumption of a particular device. These state profiles are expressed using tasks with activity profiles.

```
    task with_schedsleep {
        t_sched = parameter {
            default = 2h;
            min = 0;
            max = 4h;
        }
        TEC = activtity-profile {
            {
                requires = tv_watch;
                length = 14h;
            },
            {
                requires = tv_sleep;
                length = 10h - t_sched;
            },
            {
                requires = tv_schedsleep;
                length = t_sched;
```

```
            }
            iterations = 365;
        }
    }
```

The example above shows the state profile for devices with a scheduled sleep state. The UHA activity profile shown includes three segments, one for each of the states, along with the expected duration. In this case, the duration of the sleep and schedsleep segements depends on the configuration parameter “t_sched,” which defaults to 2 hours.

PC High-Level Modeling

Modeling the Power States

Each power state defined in the ENERGY STAR®-specification is represented by a UHA state.

```
    state active {
        power = TBD;
        transition {
            from = &sleep;
            latency = 5s;
        }
        transition {
            from = &off;
            latency = 60s;
        }
    }
```

As the power levels are not prescribed by the ENERGY STAR®-specification, the power levels are left undefined using the “TBD” construct.

The transition latencies from the other power states are specified explicitly inside UHA transitions.

Even without explicitly defined power levels, it is possible to express relationships between power levels of different states using assertions.

Maximum allowed power levels for a particular state are described as in the following example:

```
state sleep {
    requires = &memory/context;
    power = TBD;
    max_power = max_sleep_power + 0.03 * C;
    assert = <power LE max_power>;
}
```

Furthermore, for each state, a corresponding task has been defined that represents the requirement for a particular device state.

Modeling the TEC Base Allowance

The TEC base allowance is described by defining the different types of devices and then asserting that only one of them may be instantiated:

```
device {
    requires = &pc_ES250/present XOR
        &pc_ES425/present XOR
        &pc_ES690/present XOR
        &notebook/present;

    pc_ES250 {
        description = "Computer with ES <= 250";
        state present { power = 50kW / 365 / 24; }
        state ignore { power = 0; }
    }
```

Using the “requires” property as well as the XOR operator, one can assert that exactly one of the listed devices must be present.

Each device has a TEC base allowance defined using the associated power model in the “present” state.

Modeling the TEC Adders

The TEC adders defined in the ENERGY STAR specification are described as UHA components. As ENERGY STAR®-does not prescribe power levels by power state for those additional features, the entire TEC adder figure is converted to a power number for the component.

TV High-Level Modeling

The TV high-level model is engineered based on Samsung UN65KU7000F (Samsung Electronics, 2016).

The states the research team modeled for the ENERGY STAR-certified TV includes all the states mentioned in the ENERGY STAR®-specification. These states and associated dependencies are in the table below.

Table 2: States in TV High-Level Modeling

State	Dependencies	Description
On	Screen on, CPU and other components on	The TV is connected to the main power and is capable of producing dynamic video.
Passive Standby	Screen off, CPU in S1 state	The TV is inactive and can be switched into another mode with only the remote-control unit or an internal signal.
Active Standby / Low	Screen off	The TV is inactive, can be switched into another mode with the remote-control unit or an internal signal, and can be switched into another mode with an external signal.
Active Standby / High	Screen off	The TV is inactive but is exchanging/receiving data with/from an external source and can be switched into another mode with the remote-control unit, an internal signal, or an external signal.
Download Acquisition Mode	Screen off, Ethernet on	The TV is inactive but is downloading data from the network
Off	N.A.	The TV is switched off.

Source: Aggios, Inc.

The details of the model are not listed here due to space considerations. The most critical observation from this high-level modeling is the state of the screen is essential in determining the state of the TV. This observation is not surprising because when the user looks at an active screen, the user always perceives that TV is active. This fact indicates a potential opportunity for power savings because it is possible to keep the screen active while keeping other components in an inactive mode periodically for power conservations.

Gaming Console High-Level Modeling

For AGGIOS' game console prototype, the research team conducted a high-level modeling effort based on the power behaviors of Microsoft's® Xbox 360 (Microsoft, 2013). While the Xbox 1 is current as of this writing, detailed researches on the internals of Xbox 1 are unavailable from open sources at the time of the study; so the 360 was chosen as the device source.

Statistics on the power behavior of the Xbox 360's is obtained from the National Resource Defense Council's 2014 report (Delforge & Horowitz, 2014), funded by a grant from the U.S. Environmental Protection Agency. Testing was conducted by Pierre Delforge, an NRDC researcher working on electricity consumption in the IT and consumer electronics sectors and the former lead energy and climate strategist for Hewlett-Packard's sustainability group. The protocols used for NRDC's tests are described in the AGGIOS did no separate testing of different 360 models.

Subsystem Energy Modeling

With the power profile derived from the high-level modeling performed earlier, the research team modeled the proposed virtual prototypes. The detailed modeling gives the engineers opportunities to experiment on different hardware options before developers of a plug-load device build the hardware; therefore, the modeling allows the engineers to make compromises between power and other design factors, such as performance, cost, and form factor.

Modeling Techniques

It is often impractical to measure directly the power consumption of a component when the component is already soldered on a printed circuit board (PCB) due to the difficulties to inserting shunt registers into the voltage delivery network in an existing PCB. Requiring measuring the power before building a PCB may also be impractical in some cases when the component requires complex logic and auxiliary components to drive it and, therefore, will not enter the desired state when powered individually.

Three techniques have been used to build power/latency models of components: datasheet, black box, and power behavior analysis. For each component, the team chose one of these techniques based on the available data and accessibility of the component.

Modeling From Datasheet

Some components have published the respective operating states, power consumption, and latency values in the associated data sheets. Sometimes, these values can be used for modeling.

An example for such a model is the USB controller on the ZCU100 board, as shown in **Error! Reference source not found.**.

Figure 4: A Table Listing Typical Power Consumption Values From a USB Controller IC (Microchip, 2017)

TABLE 9-3: DEVICE POWER CONSUMPTION

	Typical (mA)		Typical Power
	VDD33	VDD12	(mW)
Reset	0.8	1.8	4.8
No VBUS	2.0	5.0	12.6
Global Suspend	2.0	5.2	12.9
4 FS Ports	39	35	170
4 HS Ports	53	42	222
4 SS Ports	55	683	1001
4 SS/HS Ports	93	688	1132

Source: Aggios, Inc.

In other cases, the data from the datasheet may need to be calibrated before using them for modeling. First, the power or current values given by the datasheet may be a maximum value instead of an average value, as shown in the example in **Error! Reference source not found.**. Second, the data given by the datasheet may contain excessive details. For example, an LPDDR4 component contains 15 states detailing the average current draws for operation details such as burst read and burst write. However, directly using these data is challenging because it requires modeling detailed memory access patents from the application software and system software.

Figure 5: Power Model of N25Q128A

Parameter	Symbol	Test Conditions	Min	Max	Unit
Input leakage current	I_{LI}		–	±2	µA
Output leakage current	I_{LO}		–	±2	µA
Standby current	I_{CC1}	$S = V_{CC}$, $V_{IN} = V_{SS}$ or V_{CC}	–	100	µA
Deep power-down current	I_{CC2}	$S = V_{CC}$, $V_{IN} = V_{SS}$ or V_{CC}	–	10	µA
Operating current (fast-read extended I/O)	I_{CC3}	$C = 0.1V_{CC}/0.9V_{CC}$ at 108 MHz, DQ1 = open	–	15	mA
		$C = 0.1V_{CC}/0.9V_{CC}$ at 54 MHz, DQ1 = open	–	6	mA
Operating current (fast-read dual I/O)		$C = 0.1V_{CC}/0.9V_{CC}$ at 108 MHz	–	18	mA
Operating current (fast-read quad I/O)		$C = 0.1V_{CC}/0.9V_{CC}$ at 108 MHz	–	20	mA
Operating current (program)	I_{CC4}	$S\# = V_{CC}$	–	20	mA
Operating current (write status register)	I_{CC5}	$S\# = V_{CC}$	–	20	mA
Operating current (erase)	I_{CC6}	$S\# = V_{CC}$	–	20	mA

Source: Aggios, Inc.

For the first case, it is possible to adjust the power models using measured data. It is often safe to assume that when the component is at the lowest power state, there is only a negligible difference between the power value given from the datasheet and the measured average power.

The reason is simply the power of the lowest state is often very close to zero. In the example in **Error! Reference source not found.**, the maximum power of the lowest state is only 18 microwatts (µW) (10µA current draw from a 1.8 volt [V] voltage supply). Because the total power consumption of the virtual prototype is much higher (100W~200W), directly using this number incurs only a negligible error, which is only 18µW in the worst case. The power models of other states can then be calibrated by measuring the power consumed by the whole board while component is at the desired state, measuring the power consumed by the whole board while the component is at the lowest state, and taking the difference value.

For the second case, it is possible to combine the power consumptions of the detailed states into a smaller number of states by making assumptions on a usage paten, so that many states can be combined into a smaller number of superstates.

Modeling by Direct Measurements

When a component is powered by dedicated power cables, it is sometimes possible to build the power model by direct measurements.

Power Behavior Analysis

In some cases, it is difficult to measure the power consumption of some components. For example, the component may be sharing voltage sources with other components. In these cases, the power model was built by measuring the total power consumption of the television under different usage scenarios and comparing the results by analyzing the usages of different components under these usage scenarios.

Power Models of Components

Common Components

All four of the virtual prototypes are modeled based on the ZCU102 platform (Xilinx, 2017). Some components are common. The core of the ZCU102 platform is a system-on-chip (SoC) named Zynq UltraScale+ MPSoC (ZynqMP), which is a heterogeneous multicore SoC containing a powerful quad-core ARM Cortex-A53 suitable for running a complex operating system such as Linux or Windows, a dual-core, real-time processor, a large programmable logic component, or a field programmable gate array (FPGA) The integration of the FPGA provides the research team the ability to implement digital components that are not directly available on the board, e.g., a digital TV tuner that decodes digital TV signals received in an analog format from the physical layer.

Processors

All four virtual prototypes developed in this project contain software that require processors. For example, in the real world virtually every TV, set-top box, gaming console, and personal computer (PC) relies on an operating system that needs to run on a processor.

The team's virtual prototypes contain processors modeled after the power, performance, and latency behaviors of the SoC, which contains 4 ARM Cortex-A53 (Arm Limited, 2016) cores. The cores use a multilayer cache system that sizes 32/32/1 megabyte (MB).

The power models of these on-chip components were produced with the help from Xilinx, the manufacturer of the SoC.

Flash

The power profile of the flash memory on the board has been modeled based on the reference manual of the Micron N25Q128A (Micron, 2012) Table 3 outlines the operating modes.

Table 3: Power States of the Flash Chip

Mode	Current (Symbol)	Voltage (V_{CC})
Standby	100 uA (I_{CC1})	1.7V to 2.0V
Deep power-down	10 uA (I_{CC2})	1.7V to 2.0V
Operating / fast-read extended I/O	15 mA (I_{CC3})	1.7V to 2.0V
Operating / fast-read dual I/O	18 mA (I_{CC3})	1.7V to 2.0V
Operating / fast-read quad I/O	20 mA (I_{CC3})	1.7V to 2.0V
Operating (write status register)	20 mA (I_{CC5})	1.7V to 2.0V
Operating / erase	20 mA (I_{CC6})	1.7V to 2.0V

Source: Aggios, Inc.

The power-up latency for this chip is 150 us.

Double Data Rate (DDR)

The DDR chip used in the ZCU100 board is an LPDDR4 module with no documents publicly available. Therefore, the model has been based on the similar MT53B384M64D4 from the same manufacturer (Micron).

The power model of a DDR depends on many operations performed on the DDR. For example, the reading operation from the chip, writing operating to the chip, and keeping the chip in self-refresh without any reading or writing consume different amounts of current from the three voltage sources. However, directly using the models with such an excessive amount of details will require modeling the detailed behavior of the memory access patterns of the software, which is typically impractical.

To simplify the model while still having the level of accuracy suitable for power modeling in a television prototype, the 15 states mentioned in the datasheet have been condensed into 3 states: operating, power_down_referesh, and standby. All the states describing the short-term behaviors, such as active-precharge and burst-refresh, are ignored. This is because the DDR stays in these states only for a short amount of time, and the power in these states contribute a

negligible part to the total energy consumption of the TV prototype. All the active states, including the multiple reads, writes, and refresh states are combined into an active state by assuming the chip spends equal amount of time in read, write, and all_bank_refresh. The error caused by this simplification is in the range of [-0.33W, 0.29W].

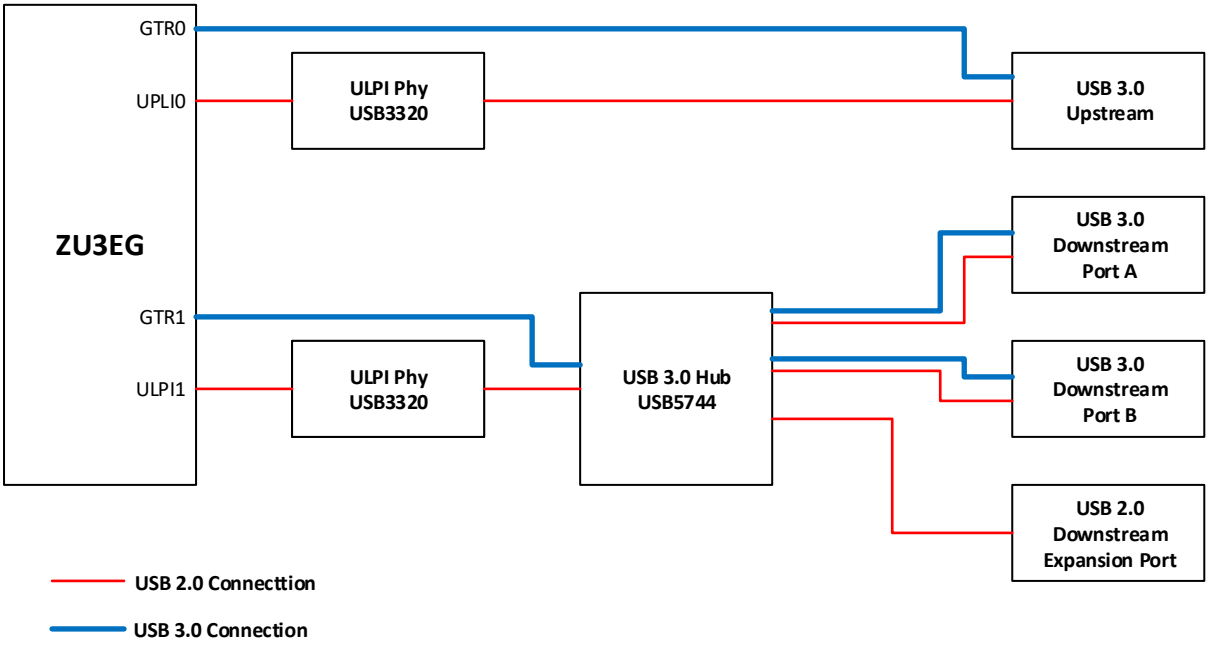
The power consumed by referencing the component is related to the temperature because this DDR supports Auto Temperature Compensated Refresh. When overheated (e.g., 85˚C), the power consumption is significantly more than normal. However, the detailed power curve from a low temperature to a high temperature is not available. Therefore, the temperature behavior has been modeled as an overheat factor in the power_down_refresh state:

```
overheat = parameter {
        values = 0, 1;
        default = 0;
}
state power_down_referesh {
        requires = &ddrc/on;  //requiers the DDR controller
        power = 0.4mA * vdd1 + 1.5mA * vdd2 + 0.02mA * vddq +
                 overheat * ( 1.4mA * vdd1 + 4.7mA * vdd2 + 0.02mA * vddq );
}
```

USB Host

The USB host device on the board has been modeled using the setup as shown in the figure.

Figure 6: USB Host Setup



Source: Aggios, Inc.

This setup uses two types of devices: the USB ULPI transceiver USB3320 and the USB 3.0 Hub USB5744. The respective power models are listed as follows.

ULPI Transceiver

The ULPI transceiver is modeled based on USB3320.

Table 4: ULPI Power States

State	Current	Voltage
Synchronous Mode, Default Configuration	28.0 mA	1.8 ± 0.2V
Synchronous Mode, HS Operation with active USB transfer	29.4 mA	1.8 ± 0.2V
Synchronous Mode, FS/LS Operation with active USB transfer	22.5 mA	1.8 ± 0.2V
Serial Mode FS/LS USB	2.4 mA	1.8 ± 0.2V
UART Mode	2.4 mA	1.8 ± 0.2V

Low-Power Mode	0.7 uA + 30 uA	1.8 ± 0.2V
Standby Mode	0.6 uA + 0.1 uA	1.8 ± 0.2V

Source: Aggios, Inc.

The latency values for the components to switch from the low-power mode ranges from 0.52 milliseconds (ms) to 2.24 ms, depending the reference clock. The value of 0.52 ms has been adopted because it is possible to set the reference clock to the desired frequency for a shorter latency.

USB 3.0 HUB

The USB 3.0 HUB is modeled based on USB5744.

State	Power
Reset	4.8 milliwatt (mW)
No VBUS	12.6 mW
Global Suspend	12.9 mW
4 FS Ports	170 mW
4 HS Ports	222 mW
4 SS Ports	1001 mW
4 SS/HS ports	1132 mW

Considering all the steps required for the starting up of this device, the latency for it to transfer from the global suspend state to the active states has been modeled as 1 ms.

Wi-Fi and Bluetooth

The Wi-Fi and Bluetooth module has been modeled based on the specification of TI WL1831MODE. The module is powered by a 1.8 V source.

The power consumption of the Wi-Fi part is complex as it depends on the operations made to the module (transmitting or receiving) and on the modes that the module uses for transmitting or receiving, which in turn depends on the environment where the module is used. To avoid modeling too much details on the environment, the Wi-Fi module has been simplified into four states: read_write, mostly_read, mostly_write, and inactive. It has been assumed that the mostly_read state consumes the average power of all read modes of the Wi-Fi receiver. Likewise, the mostly_write state is assumed to consume the average power of all write modes of the Wi-Fi transmitter. The read_write state consumes the sum of the power of the mostly_read and mostly_write states.

Wi-Fi *Receiver*

Table 5l Wi-Fi Receiver States

State	Current
LPM 2.4GHz RX SIDO20 single chain	49 milliamps (mA)
2.4GHz RX search SIDO20	54 mA
2.4GHz RX search MIMO20	74 mA
2.4GHz RX search SISO40	59 mA
2.4GHz RX 20M SISO 11 CCK	56 mA
2.4GHz RX 20M SISO 6 OFDM	61 mA
2.4GHz RX 20M SISO MCS7	65 mA
2.4GHz RX 20M MRC 1 DSSS	74 mA
2.4GHz RX 20M MRC 6 IFDM	81 mA
2.4GHz RX 20M MRC 54 OFDM	85 mA
2.4GHz RX 40M MCS7	77 mA

Source: Aggios, Inc.

To simplify the power model, the average current draws from the modes in the table have been used, amounting to 66.82 mA.

Wi-Fi Transmitter

Table 6: Wi-Fi Transmitter States

State	Current
2.4GHz TX 20M SISO 6 OFDM 15.4 dBm	285 mA
2.4GHz TX 20M SISO 11 CCK 15.4 dBm	273 mA
2.4GHz TX 20M SISO 54 OFDM 12.7 dBm	247 mA
2.4GHz TX 20M SISO MCS7 11.2 dBm	238 mA
2.4GHz TX 20M MIMO MCS15 11.2 dBm	420 mA
2.4GHz TX 40M SISO MCS7 8.2 dBm	243 mA

Source: Aggios, Inc.

Bluetooth BR and EDR

The Bluetooth module also has a power consumption based on the transfer modes determined by the environment. Similar to the method used for the Wi-Fi model, the average power in all the modes has been used as the active power of the Bluetooth module.

State	Current
Full throughput ACL RX: RX-2DH5	18.0 mA
Full throughput BR ACL TX: TX-DH5	50.0 mA
Full throughput EDR ACL TX: TX-2DH5	33.0 mA
Other modes	Not modeled as they are not used in the intended use cases.

Source: Aggios, Inc.

The average current draw in the three active modes are 35.3mA.

The latency of the Wi-Fi/BT module is modeled as 5 ms.

TV Components

Besides the common components, the TV virtual prototype also contains several unique components.

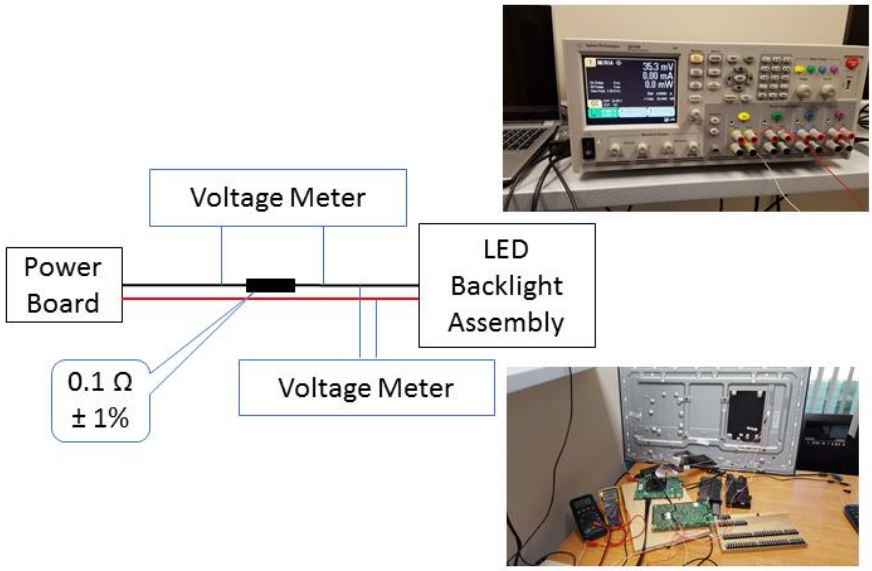
LED Backlight

The power model of the light-emitting diode (LED) backlight in the virtual prototype is based on the LED backlights used in Samsung 43KU7000 TV.

The backlight assembly is powered by two pairs of cables from the power board. When connected to the workload, the voltage and current vary based on the backlight level that is chosen by the user from the menus of the television.

For this project, the team derived two LED backlight models, with or without ambient brightness control (ABC).

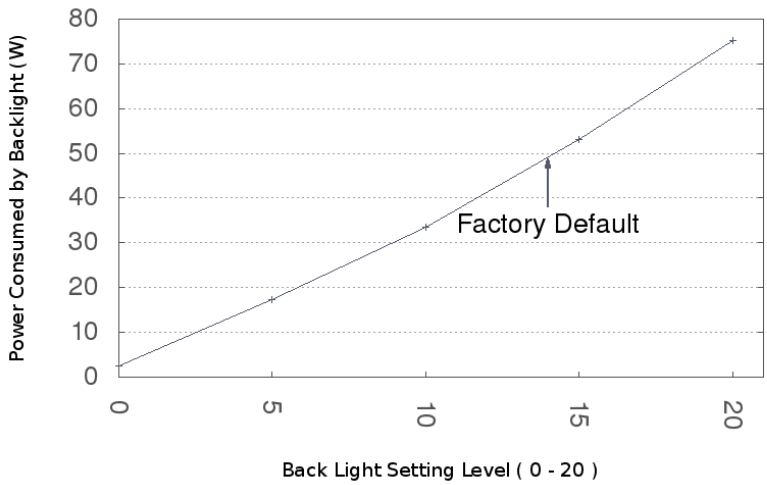
Figure 7: Measurement Setup for Building the Power Model of the LED Backlight



Source: Aggios, Inc.

The setup shown in **Error! Reference source not found.** has been used for measurements. A shunt resister is added onto the cable that powers the LED backlight assembly. A voltage meter is used to measure the voltage drop on the resister, which is then used to calculate the current flowing into the backlights. The voltage added to the backlight assembly is measured at the same time to calculate the power consumption. The settings of the backlight level are changed using the menu on the television and derived a power model, as shown in **Error! Reference source not found.**.

Figure 8: Back Light Power Model



Source: Aggios, Inc.

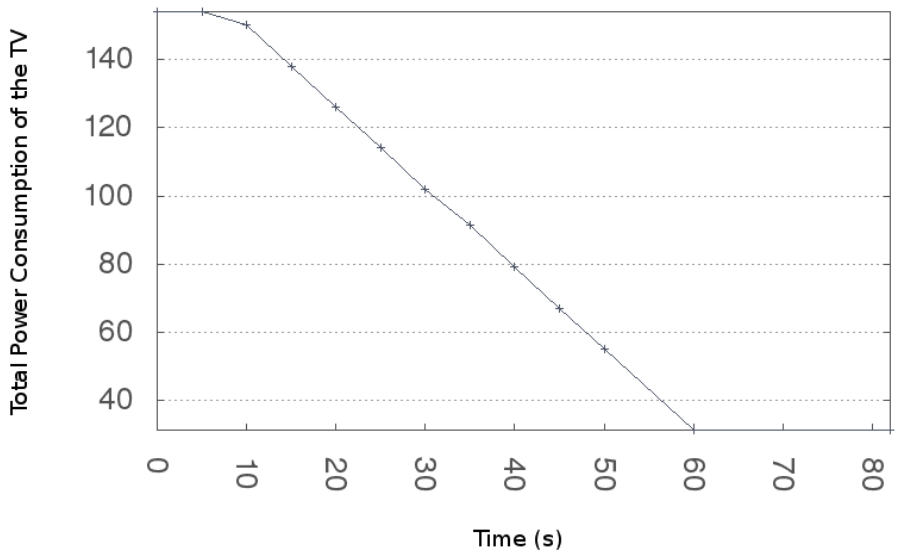
To simplify the modeling and calculation, a linear model is used to approximate the curve in **Error! Reference source not found.** as:

$$P = 3.6006\,s - 0.0187$$

where P is the power consumption of the back-light assembly, s is the back-light level setting. This linearized model is very close to the measured curve, with an R^2 value of 0.9919.

Many modern televisions, including the Samsung TVs that have been measured in the lab, contain ambient brightness control, which reduces the brightness of the television in dark environments for energy savings.

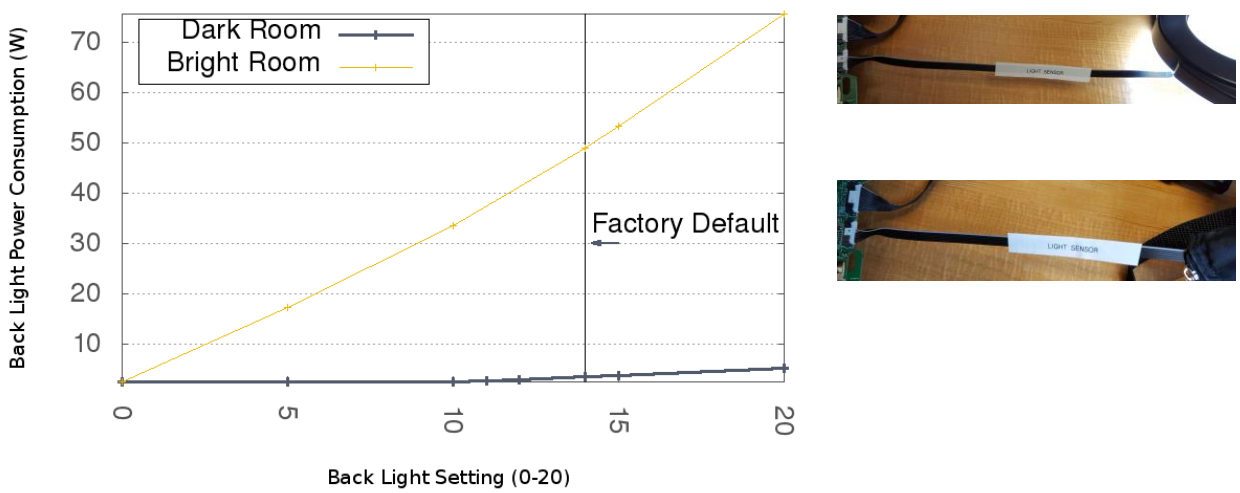
Figure 9: TV Power Drops After a Sudden Move to a Completely Dark Environment



Source: Aggios, Inc.

The following experiments have been performed. A Samsung 65KU7000 TV was used with the backlight set to 20, which is the maximum value. A relatively bright video has been chosen for playback, while the speaker was muted. The backlight sensor is covered to simulate the scenario that the environment becomes completely dark. The change in power consumption of the television is plotted in **Error! Reference source not found.**. Time 0 is the time when the light sensor is covered. For about 5 seconds, the total power did not have a significant change, indicating that the television did not react to the change in the ambient light level during this time. After that, the total power drops linearly until it reaches the minimum time of 60 seconds.

Figure 10: Back Light Power Consumption in Dark or Bright Rooms



Source: Aggios, Inc.

Based on the observation made in these experiments, a linear model has been adopted for the back light with active brightness control as:

$$P = \begin{cases} P_{\max} & m \cdot s \geq \dfrac{P_{\max} - \beta}{\alpha} \\ \alpha \cdot m \cdot s + \beta & other \\ P_{\min} & m \cdot s \leq \dfrac{P_{\min} - \beta}{\alpha} \end{cases}$$

where m is the level of ambient light detected, where the value is assumed to range from 0 to 500; s is the back light setting from 0 to 20. Other symbols are constant parameters.

The parameters used in the model are calibrated as follows. The power consumption of the backlight is measured under different back light settings in a dark room and a bright room, as shown in Figure 10. A dark room is emulated by covering the ambient light sensor, while the bright room is emulated by putting the ambient light sensor under a strong light.

$$P = \begin{cases} 75.68 & m \cdot s \geq 10000 \\ 0.007343 \cdot m \cdot s + 2.25 & other \\ 2.5 & m \cdot s \leq 34 \end{cases}$$

Screen

The screen modeled is a conventional liquid crystal display (LCD) system. Such LCD systems consume almost constant power regardless of the display content (Dong, Choi, & Zhong, 2009). Therefore, the screen is simply modeled with two states—active and inactive—with the active state consuming a constant power and the inactive state having a power consumption of 0.

The constant power consumption of the active state has been derived based on several experiments as follows. The first step is to find a mode with the lowest total power consumption while keeping the screen active, which consumes 18.88 W. The next step is to measure the power consumption of the television with the voltage cords to the screen disconnected, which is 10.2W. The difference between the two modes is assumed to be the constant power consumption of the active state, which is 8.68 W.

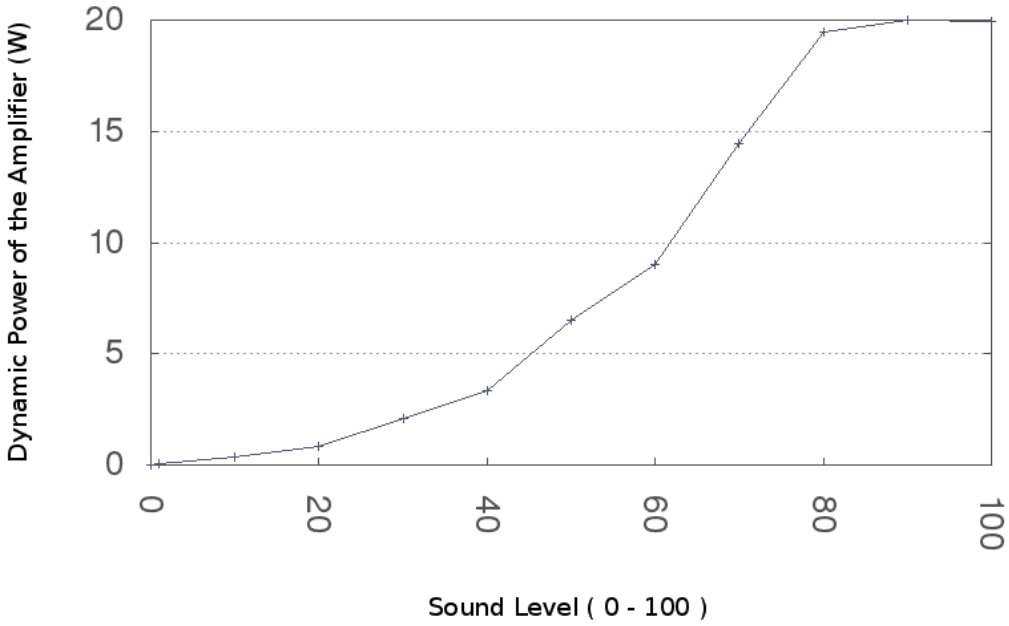
Speaker and Amplifier

The amplifier has been modeled based on TI TAS5706, the audio amplifier used in the Samsung TV. The amplifier consumes a small amount of power when it is in the power-down mode. When it is active, it consumes static power and dynamic power based on the sound it emits.

The static power of the amplifier is calculated from the datasheet, while the dynamic power of the amplifier is derived from an experiment by playing a 1 kilohertz (kHZ) test sound while keeping other operating conditions constant, and measuring the total power consumption of the TV.

The power model derived is plotted in **Error! Reference source not found.**.

Figure 11: Power Model of the Amplifier



Source: Aggios, Inc.

A parameter sound level is used when modeling the power of the amplifier. This parameter is not identical to the sound level setting that the user selects via the remote control. Because normal audio signals changes by time, the related instant power consumption also changes. When modeling the power consumption of the television virtual prototype, the parameter of sound level should be interpreted as the average sound level of the system during a period.

Table 7: States of the Amplifier Driving the Speaker

State	Power
Active	33 mA * 3.3V + Dynamic Power in **Error! Reference source not found.**.
Power Down	58 µA * 3.3V

Source: Aggios, Inc.

The latency to switch from the power down state to the active state is 120 ms.

HDMI Host Subsystem

Due to the lack of documents and internal knowledge, the model of the HDMI subsystem is inferred from the observations made in a list of experiments by playing a blank (all dark) video in different ways.

Table 8: Power Models of the HDMI Host

Experiment Number	Description	Total Power Consumption
1	Play blank video with the built-in Youtube App without an HDMI cable plugged	18.88 W
2	Play blank video with the built-in Youtube App with an HDMI cable plugged	20.83 W
3	Play black video using the laptop via HDMI	25.70 W
4	Show black screen using a laptop via HDMI	25.67 W

Source: Aggios, Inc.

The differences in power between Experiments 1 and 2 indicate that the HDMI subsystem consumes a certain amount of power even when there is no signal being transmitted. The research team assumes that this power is required to build a physical connection.

The active power of the HDMI subsystem is using the differences in power between Experiments 2 and 3. This value is an overestimation because there are some components used in Experiment 2 but not used in Experiment 3. The research team can account for one of them, the Wi-Fi subsystem, using the model presented earlier. There may be other components that are uncounted for. However, the team assumes that the power consumed by those components is insignificant because similar video decoding tasks run well on low-power devices, e.g., an Apple TV consuming only 2 W as measured by the research team.

In conclusion, the HDMI subsystem is modeled as:

```
HDMI {

    state plugged {

        state active {

            power = 25.70W - 20.83W + 49mA * 1.8V;

        }

        state inactive {

            power = 20.83W - 18.88W;

        }

    }
```

```
    state unplugged {
        power = 0;
    }
}
```

TV Tuner

The TV tuner is assumed active only when the TV is in use. A simple method is used to model the TV tuner from two measurements.

Measurement 1: Using the built-in YouTube app to play a blank (all-dark) video with the backlight turned to the lowest level and get a power measurement of 18.88 W.

Measurement 2: Keep the backlight at the lowest level, turn to the TV mode with an antenna connected, play a program with a dark picture, and observe the lowest power consumption during play as 21.91 W.

The difference between the two measurements is taken as the active power of the TV tuner. This number may be inaccurate because:

1. The lowest power consumption during TV play may still an image, which causes the screen to consume more power than the all-dark video in Measurement 1.
2. The YouTube app may use some components that are not used during Measurement 2.

With the understanding of the potential modeling errors, the result related to the TV tuner should be interpreted carefully. However, this error may be insignificant when using the television virtual prototype to simulate common usage scenarios in California households because TV tuners are used less often. According to VIZIO, “Most households commonly stream or watch live TV from cable and satellite boxes. Neither of these requires a tuner. ” (VIZIO Inc, 2018)

Gaming Console Components

Besides the common components, the gaming console virtual prototype also includes the following special components.

External Power Supply

The gaming console that has been modeled in this virtual prototype uses an external power supply with three states—passive_draw, low_power, and max_power.

```
    external_power_supply {
       high = capability {type = shared;}
       medium = capability {type = shared;}
       low = capability {type = shared;}
       state passive_draw {power=20mW;
               capabilities = low;}
       state low_power{power=400mW;
               capabilities = medium, low;}
       state max_power{power=6W;
               capabilities = high, medium, low;}
  }
```

Peripheral Component Interconnect Express (PCIE) Devices

The gaming console modeled in this virtual prototype contains several PCIE devices, including a Wi-Fi board, an RF module that is used to communicate with external hand-held controllers, and an external graphics card.

```
pcie_devices {
        wifi_board {
           skyworks_amp {
               state on {power = 600mW;}
               state off {}
           }

           cet_SPDT_switch {
               state on {power = 600mW;}
               state off {}
           }

           marvell_soc {
               state on {power = 600mW;}
               state off {}
           }

           state on {
               requires = &skyworks_amp/on, &cet_SPDT_switch/on,
&marvell_soc/on;
           }
           state off {}
        }
        rf_module_x85 {
           state on {power = 300mW;}
           state off {}
        }

        external_graphics_card {
```

```
            voltage-parent = &vdd_gt;

            voltage-supply vdd_gt {
                voltage-parent = &vdd_12V;
                voltage-output = capability {
                    max = 12V;
                }
            }
            state on {
              voltage-state = ON;
              state full_load {power=68.4W;
                                requires = &graphical_accelerator/active,
                                  &external_power_supply/high;}
              state desktop_load {power=31.3W;}
              state passthrough_load {power = 2W;
                                  requires = &external_power_supply/medium;}
            }
            state off {voltage-state = OFF;}
        }
    }
```

Optical Drive

Today, optical drives are still used widely in gaming consoles, though the usage is likely to drop in the near future as Xbox has announced plans to release a new product without optical drives in 2019. A typical installed optical drive on a device consumes a certain amount of power even when it is not reading an optical disk. For these reasons, the optical drive is modeled only considering in the on and off states.

```
optical_drive {
          state on {
            power = 2W;
          }

          state off {}
         }
}
```

Hard Drive

The virtual prototype includes a model of an internal hard drive based an existing work (Bianchini, 2004).

```
      hard_drive {
          description = "2TB HDD WD Green - WD20EZRX";

          voltage-parent = &hdd_v12;

          state high_speed {
              voltage-state = ON;
              state high_speed_readwrite {
                  power = 3590mW;
                  transition {
                      from = &down;
                      latency = 5300ms;
                  }
```

```
            }
            state high_speed_idle {
                power = 3400mW;
            }
        }
        state low_speed {
            voltage-state = ON;
            state low_speed_readwrite {
                power = 3250mW;
                transition {
                    from = &down;
                    latency = 3500ms;
                }
            }
            state low_speed_idle {
                power = 3100mW;
            }
        }
        state down {
            power = 590mW;
            voltage-state = OFF;
        }
    }
}
```

Set-Top-Box Components

The set-top-box virtual prototype uses most of the components from the common components, with an important addition—the DOCSIS modem. While other functionalities of a set-top box can be implemented on the ZCU100 board directly, the board itself does not contain a device that can be used for communications via cable networks.

```
docsis3 {

    state active {

        power = 3.8W;

        transition {

            from = &inactive;

            latency = 250ms;

        }

    }

    state inactive {

        power = 0.3W;

    }

}
```

PC Components

All PC functionalities can be implemented on the built-in components of the ZCU100 board, and, therefore, no additional components were added to the model.

Latency Modeling

This project uses two methods to model the latencies of any components: datasheet and logging time stamps. These methods can be used individually and can be used together to estimate upper bounds and lower bounds of the state transition latency of a component.

After having estimates on the state transition latencies of each component, the state transition latency of the plug-load device can be accounted for by considering the states of the necessary component that support the state of the device and the algorithm/sequences that the software drivers used to perform the component's state transitions.

Latency Modeling Based on Datasheet

Many devices provide timing information on how soon they can transition from one state to another. These numbers can be used as the latency models of the components.

However, there are several key disadvantages of the latency numbers from the datasheet.

First, the latency numbers from the datasheet represent only the latency the device is designed to have. When being used as a part of a larger system, the designer usually adds a certain leeway on top of the latency number before accessing it. For example, the USB 3.0 hub used in this virtual prototype has only a state transition latency of 3.4 ms in the worst case. However, the software driver in the operating system may choose to wait 15 ms after turning on the power before performing necessary initializing of the device. This longer wait time is necessary to account for any unexpected latencies and therefore improves the overall stability of the device.

Second, the latency numbers from the datasheet are only the hardware part of the latency. Many devices require the software driver to perform initializations and calibrations before the device becomes active from the user's point of view. These software actions have to wait until the tasks get scheduled to run by the CPU.

Third, the state transition latencies of the device may not precisely be the user's perceived latencies. Sometimes, it takes a while before the user can feel the device is active after it is operational from the point of view of the software and the hardware.

Therefore, the latency models from the datasheet can be used only as a lower bound of the total latency of a component.

Latency Modeling From Logging Time Stamps

For some devices, it is feasible to print two time stamps in the logging system before and after the state transition occurs. The difference between the two timestamps represents a latency estimate of the state transitions of the component. The significant advantages of this method include:

1. The latency values estimated using this method contains the hardware and software (device driver) latencies.
2. It can be performed with minimum external devices.

However, this method may become inaccurate for several reasons. First, it is often difficult to log the exact start time of the state transition, especially when an external signal triggers the start of the state transition, e.g., a computer wakens up by a keyboard event. Second, the logging itself consumes a considerable amount of time because the log information may be written to a slow hard drive or printed through a slow serial port.

Therefore, the latency values derived from logging time stamps are considered an upper bound of the latency of the component.

User Behavior Modeling

The user's behavior greatly affects the power consumption of plug-load systems. For computers and gaming consoles, the users may be inactive or even away from the screen occasionally. The length and frequency of these inactive periods determine the ability for an aggressive power conservation controller to put the system into a lower power state. Such inactive periods can be detected by checking if the user has any input in the last period.

For TVs and set-top boxes, the users are often inactive, giving the system chances to use a lower power state. For example, the user may leave the TV on while being away from the room for an extended period. However, unlike computers and gaming consoles, the idle periods are more difficult to be detected because TVs and set-top boxes are less interactive than computers and gaming consoles. The fact that the user did not have any input for a long time via the remotes is not a good indication of inactivity because the user may be enjoying a long TV show. More accurate methods in detecting inactivity exist, e.g., via sensors. However, these are not widely in commercial products today due to cost and privacy reasons.

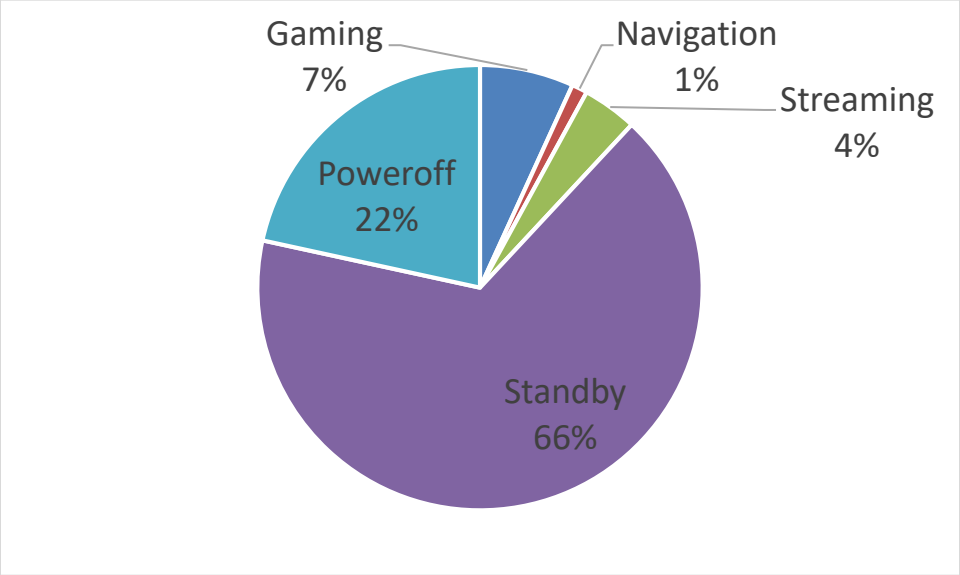
Though it is difficult to detect activities, modeling the user behavior is still essential for TVs and set-top boxes for many reasons. First, the user's preferences significantly affect the total power consumption. For example, the brightness setting of the TV'is one of the largest impact factors in the total power consumption of the TV screen. For some TV application such as USB gallery, the user's setting in duration to display a photo may give the power optimization algorithms the ability to use lower power states in some periods. Second, the user's behavior directly impact environmental factors. For example, the ambient light level is an important environmental factor that influences the TV power consumption when the TV uses adaptive lighting. This factor is significantly affected by the time in the day that the user watches TV.

Device Use Patterns

In the virtual prototype of the computers, it is assumed that a computer is under a light usage scenario where it shuts down at night has occasional usage during the daytime and stays idle between the occasional usages. This usage pattern is typical for computers in California homes and offices because people do not usually focus on their computers for an extended length of time.

The usage pattern of gaming consoles is significantly different from that of the computers. When using a gaming console, the user is usually focusing on it for a long time, e.g., for an hour. However, many users may not play it all day and all night. In addition to gaming, users frequently navigate into the gaming console for internet navigation as well, such as using the online catalog built in the gaming console UI to browse new games. Video streaming is an important use case as well. The assumed usage pattern is that of a user who uses the gaming console for three periods during a 96-hour study window. During each time window, the gaming console is used primarily for gaming, with some additional periods used for video streaming and internet navigation. **Error! Reference source not found.** plots the distributions of the time spent on gaming, navigation, streaming, standby, and power off.

Figure 12: Gaming Console Usage Distribution Used in Virtual Prototyping

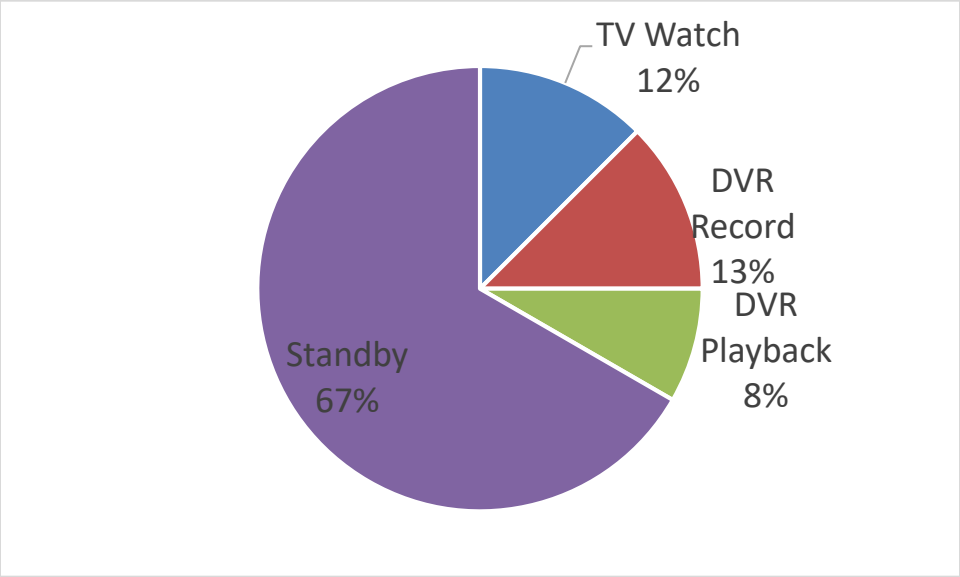


Source: Aggios, Inc.

The usage patterns of TVs are different in businesses and homes. In businesses, TVs are often kept on for extended periods with little user intervention. In contrast, at home, TVs are commonly used together with other plug-in devices, such as gaming consoles and streaming devices, and, therefore, follow the same usage patterns with the plugged devices.

The usage pattern of set-top boxes is special in that there are significant periods when the device is used for DVR recording, during which time the user does not interfere with the action of the device. **Error! Reference source not found.** describes the distribution pattern used in the set-top box virtual prototype.

Figure 13: Use Distribution in Set-Top Box Virtual Prototype

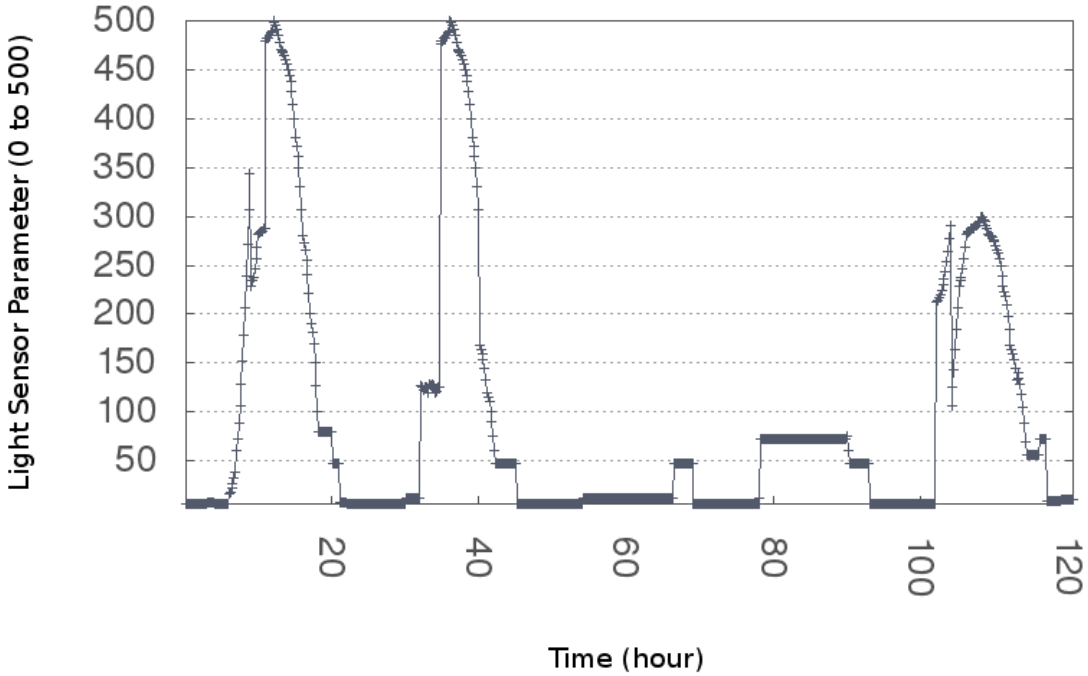


Source: Aggios, Inc.

Environment Impact Patterns

Ambient light strongly affects television power consumption and, therefore, must be modeled in the virtual prototype to achieve a high simulation accuracy. The light follows significant patterns in typical usages of TVs depending on the weather, the lighting of the room, and the user's preference settings. The lighting conditions were derived from an existing work (Ullah & Whang, 2015). **Error! Reference source not found.** lists the ambient light level that has been modeled in the virtual prototype.

Figure 14: Modeling Ambient Light Levels



Source: Aggios, Inc.

The ambient temperature has a noticeable impact to the plug load systems with active heat dissipation. When the ambient temperature is unusually high, the systems use higher energy to take away the heat by turning the fans to a faster speed.

However, in the research team's virtual prototype, the ambient temperature was not modeled for several reasons. First, plug-load devices are usually used indoors. Most California homes and offices have air-conditioning systems that addressed the differences in ambient temperatures. Second, the active cooling systems are often controlled by dedicated circuits without giving the power optimization software many controls to it for safety reasons. The ability to optimize it at runtime is, therefore, limited.

Simulation Results

The research team performs simulations with the virtual prototypes to achieve several goals.

First, for some devices, the virtual prototype reveals the energy savings from using more power efficient components while still achieving the same functional goals. These efficient components are often seen in the designs of mobile devices.

Second, for some devices, this research demonstrates the feasibility of applying more aggressive power management algorithms, which is a common practice in mobile devices.

TV Virtual Prototype

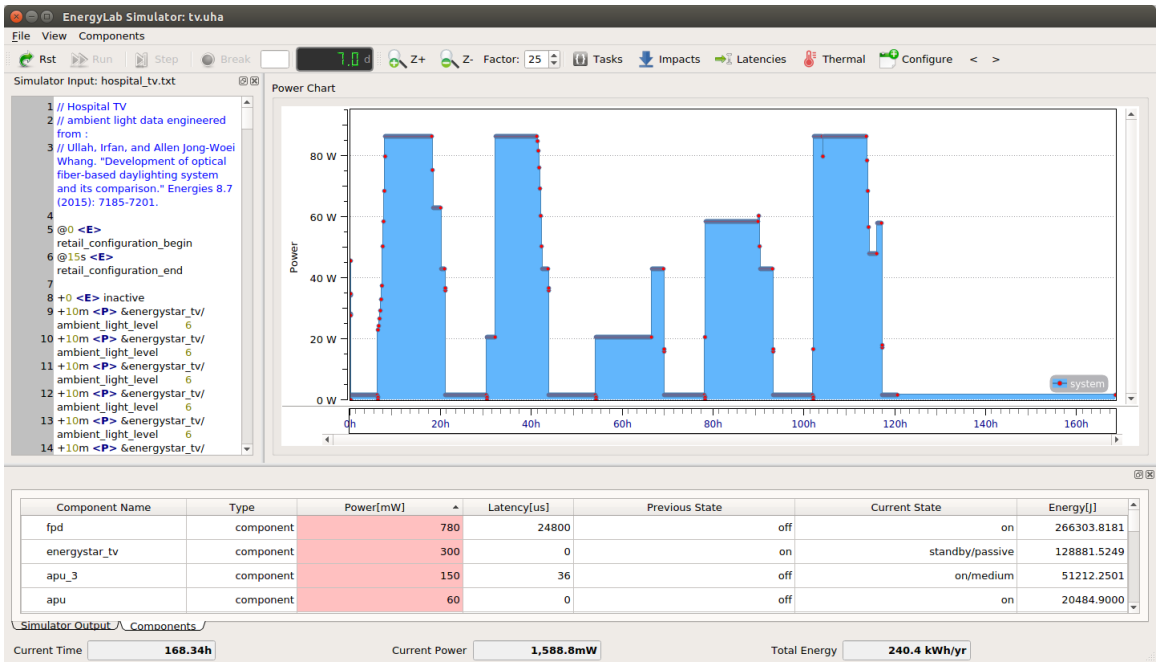
For the TV virtual prototype, this report first demonstrates the completeness of the major functionalities of the implementation by showcasing the TV in two primary use cases: the traditional TV use case and the smart TV use case. Then, this report evaluates the usage of more aggressive power management algorithms to conserve the digital part of the smart TV.

Traditional TV Use Case

The traditional TV use case uses the hospital TV simulator input file, which is not completely included here for space considerations. The usage pattern represents a typical TV used in commercial environments like a hospital. The TV is active most of the time during the day with minimum user interactions.

The power consumption results for the traditional TV use case are captured in **Error! Reference source not found.**. When comparing the power consumption curve with the curve of the ambient light levels in **Error! Reference source not found.**, the two figures follow a similar pattern. This observation indicates that in this use case, the ambient light has the greatest impact to the total energy consumption of the television. This observation can be further proven by looking at the detailed energy breakdowns in **Error! Reference source not found.**, which shows that the backlight has been consuming most of the energy in the entire simulation.

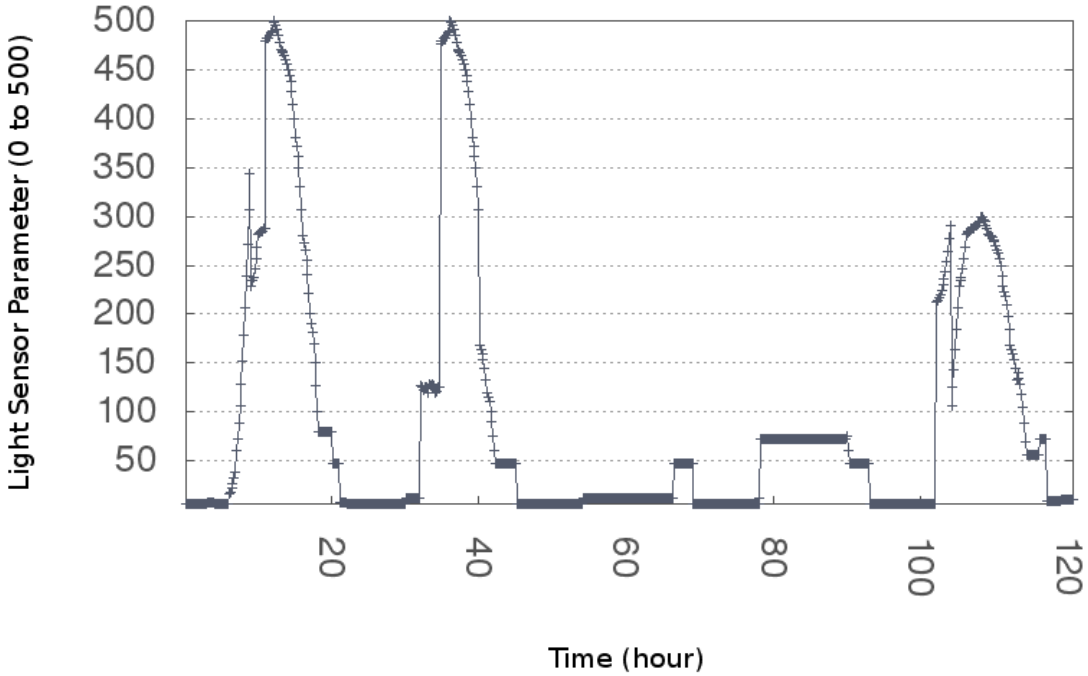
Figure 15: Power Consumption of the Virtual Prototype Under the Traditional TV Use Case



Component Name	Type	Power[mW]	Latency[us]	Previous State	Current State	Energy[J]
fpd	component	780	24800	off	on	266303.8181
energystar_tv	component	300	0	on	standby/passive	128881.5249
apu_3	component	150	36	off	on/medium	51212.2501
apu	component	60	0	off	on	20484.9000

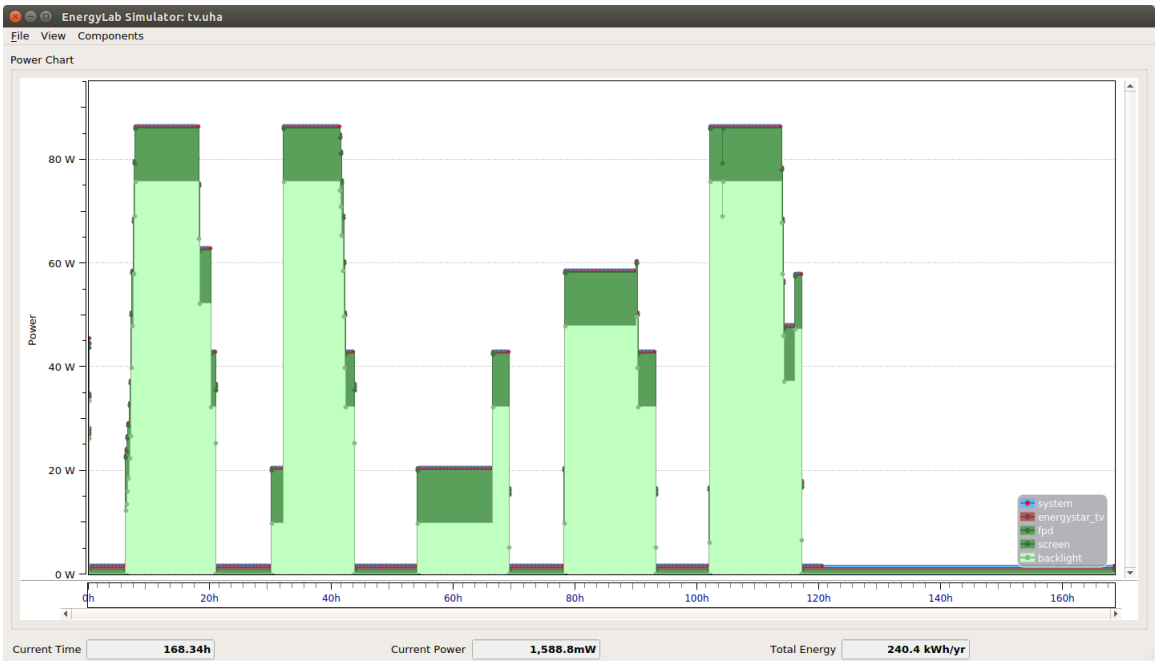
Source: Aggios, Inc.

Figure 16: Ambient Light Level in the Traditional TV Use Case



Source: Aggios, Inc.

Figure 17: Power Breakdown in the Traditional TV Use Case



Source: Aggios, Inc.

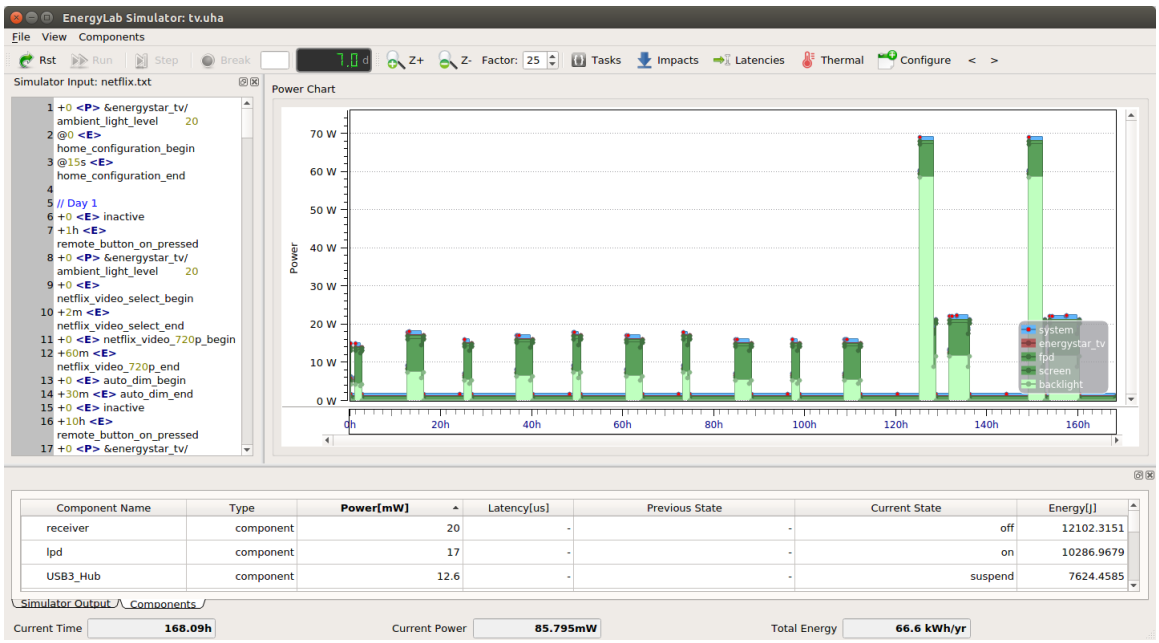
This simulation also demonstrates how television manufacturers can use a television virtual prototype to understand the energy-saving potentials of their products. The virtual prototype

reveals that more power savings can be achieved by reducing the energy used by the full-power domain (FPD) in the SoC, or by switching to a more energy efficient type of screen, such as an organic light-emitting diode (OLED) screen. Also, a virtual prototype with models calibrated with an average television usage in a region can be used to provision the total energy usage of televisions in the area due to expected weather changes.

Smart TV Use Case

The smart TV use case contains the Netflix simulator input file that simulates a typical Netflix user who streams videos online via a TV. The curve of power consumption is presented in **Error! Reference source not found.**.

Figure 18: Smart TV Use Case



Source: Aggios, Inc.

In this use case, the television spends a higher amount of time in an inactive state. Therefore, the total energy consumption is significantly lower than the traditional TV use case (240.4 kilowatt-hours [kWh]/year compared to 66.6 kWh/year).

Energy Efficiency Improvements

The work done demonstrates the potentials to improve energy efficiencies using the television virtual prototype through applying mobile technologies of the television.

Table 9: Comparisons With the Samsung TV, Excluding the Screen and the Back Light

Use Case	Samsung TV	Virtual Prototype	Power Savings	Hours / Day	Energy Savings
Traditional TV	5.8 W	5.2 W	0.7 W	4 h	1.0 kWh
Youtube	8.6 W	7.2 W	0.4 W	0.5 h	0.1 kWh
USB Gallery	7.0 W	0.1 W	7.9 W	0.5 h	1.4 kWh
Passive Standby	0.1 W	0.1 W		19 h	0 kWh
Total					2.5 kWh

Source: Aggios, Inc.

The research team compards the virtual prototype with the Samsung TV in five use cases and the results are summarized in

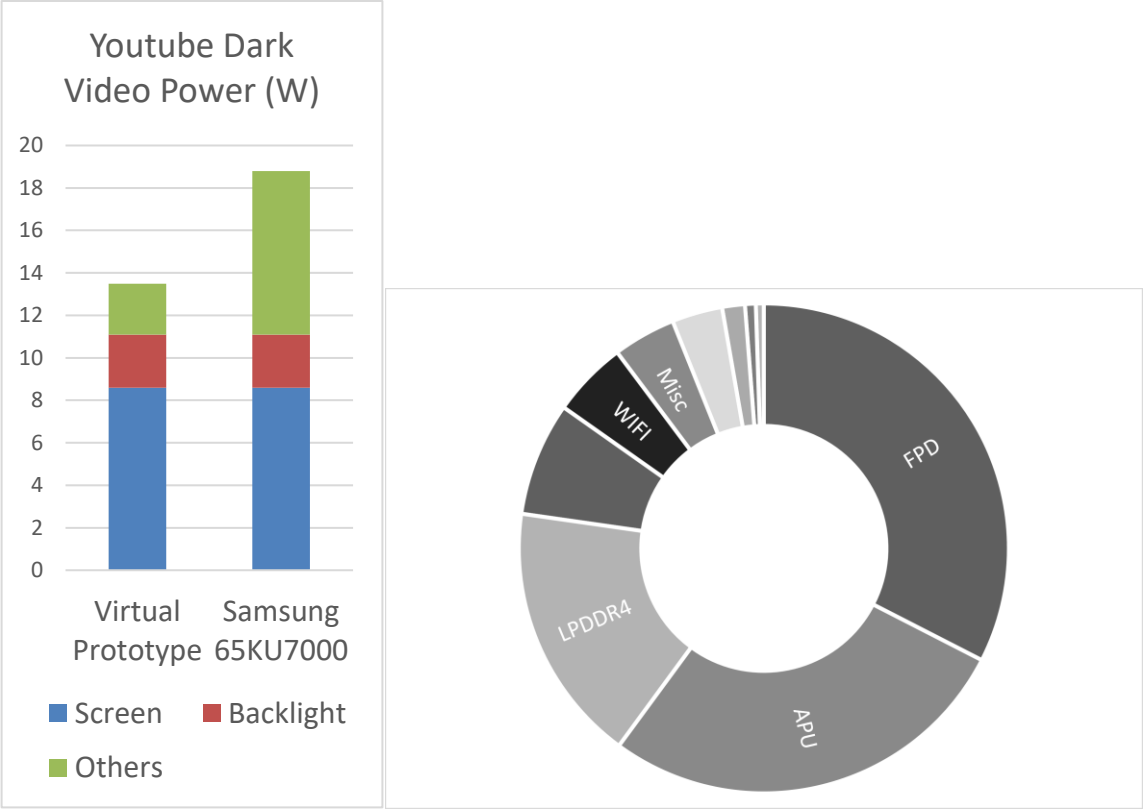
Table 9. Because the focus of this project is not on the analog part of the television, the power drawn from the screen and the backlight assembly have been excluded. In these comparisons, the least power savings come from the passive standby, in which case the Samsung TV has already optimized the energy consumption to a very low level.

The research team presents two use cases, YouTube and USB gallery, in more detail to demonstrate how the team achieved the energy savings. On the Samsung TV tested, the lowest active mode power consumption of 18.88 W has been measured when running the built-in YouTube app, playing a completely dark video, and setting all sound, light settings to the minimum. This scenario demonstrates that the power is likely to be wasted by having some unnecessary components kept on. During this experiment, the Wi-Fi activities are minimum because a compressed still video is typically tiny. Decoding such a video can be made very power-efficient because similar decoding hardware consumes only a minimum amount of energy in mobile devices. Moreover, based on estimates of the power consumed by the screen, it is likely that the remaining components except for the screen are still consuming 10.2 W of power. This amount of power is a potential opportunity for further savings because devices performing similar activities consume significantly less power.

Power minimization technologies used in mobile devices can be applied to reduce this minimum power consumption further. The concept of Android Wakelocks has been implemented in the virtual prototype to make sure that every component is turned off whenever no active requests are added to it. With this assumption, the team achieved 13.5 W of

total power consumption (**Error! Reference source not found.**), which is 28.2% saving compared with the values measured from the physical television.

Figure 19: YouTube App in the Virtual Prototype Compared With the Physical Measurement



The chart on the right side shows the power break down in the components other than the screen and the back light.

Source: Aggios, Inc.

Figure 20: Power Consumed in the USB Gallery Use Case

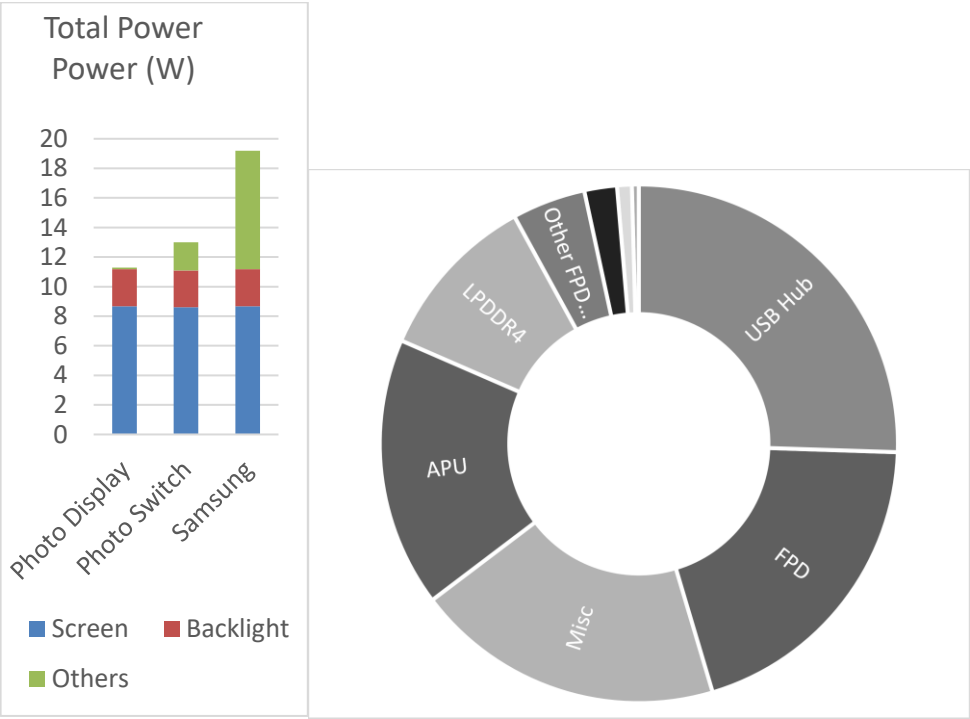


Figure on the right side shows power break down in the components other than the screen and the back light.

Source: Aggios, Inc.

Similarly, a test has been run that displays the photos on a USB drive (USB gallery). This task contains two states—one requires displaying a photo on the screen, while the other simulates the switching between one photo to another. The comparison results are available in **Error! Reference source not found.**. Compared with the YouTube case, the changes in the power breakdowns demonstrate how the virtual prototype keeps only the very necessary components active: while the YouTube case has the Wi-Fi in an active mode but keeps the USB hub inactive, the USB gallery case always keeps Wi-Fi part in a low-power mode. While in the photo switch mode, it activates the USB hub and keeps the USB hub shutdown during the photo display mode, resulting in significant additional energy savings.

Computer Virtual Prototype

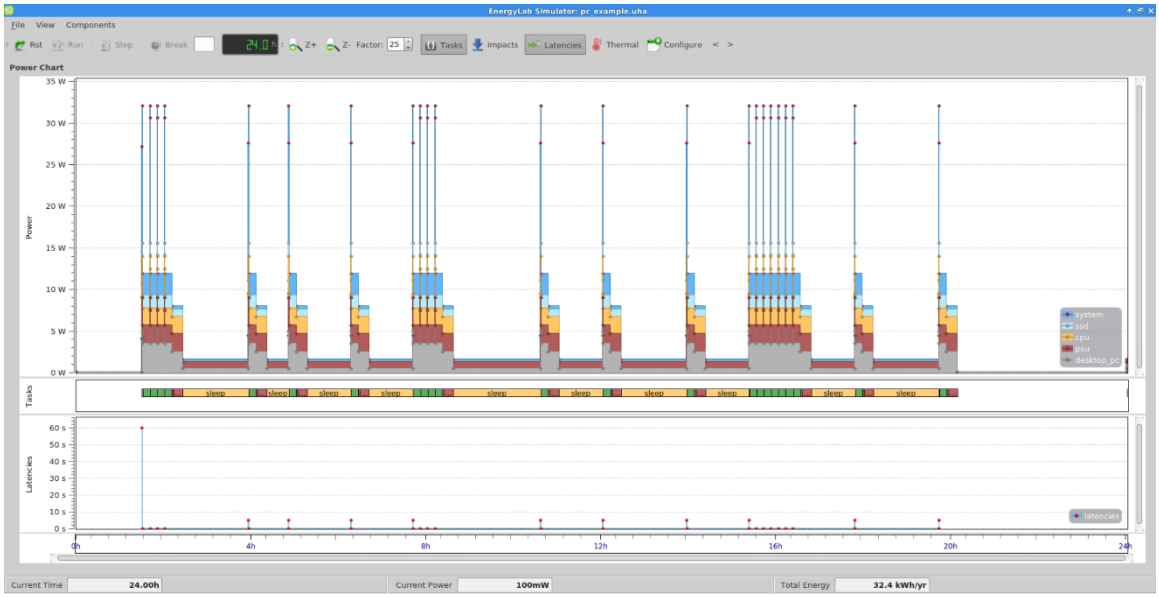
Simulations of the virtual prototype of the personal computer was performed using EnergyLab Version 0.2.001. The simulation covers the high-level example model of a PC using idealized components, as well as a detailed virtual prototype based on the Xilinx Zynq Ultrascale+ MPSoC platform.

Idealized PC Model

The high-level prototype of the PC represents an example PC using idealized components based on the breakdown of components as specified in the ENERGY STAR®-6.1 specification.

The scenario used represents a 24-hour cycle following the ENERGY STAR®-breakdown, focusing primarily on idle states.

Figure 21: Power and Latency Graph for a 24-Hour Cycle of the Ideal ENERGY STAR®-PC Model



Source: Aggios, Inc.

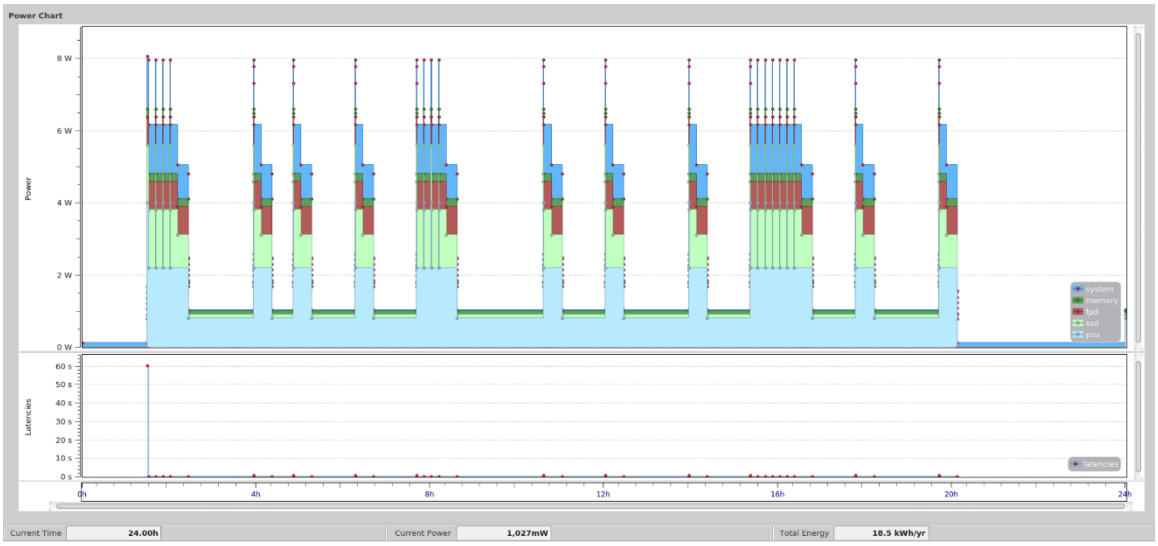
The power spikes seen in this model reflect a brief entry into the active state. However, since the ENERGY STAR®-duty cycle focuses on idle states only, those active states were limited to only 5 seconds each to minimize the impact on the resulting energy numbers.

The latency subgraph in **Error! Reference source not found.** shows a larger 60-second latency for the boot process (from off to active), while the transition from sleep to active is assumed to take 5 seconds.

Zynq UltraScale+ MPSoC-Based Model

The same stimulus used to drive the previous model was also applied to the Zynq MPSoC-based model. This model reflects the power consumption of an energy-efficient ARM-based platform; hence, the power consumption is significantly lower than in the ideal ENERGY STAR®--based model.

Figure 22: Power and Latency Graph for a 24-Hour Cycle of the Zynq MPSoC-Based Model



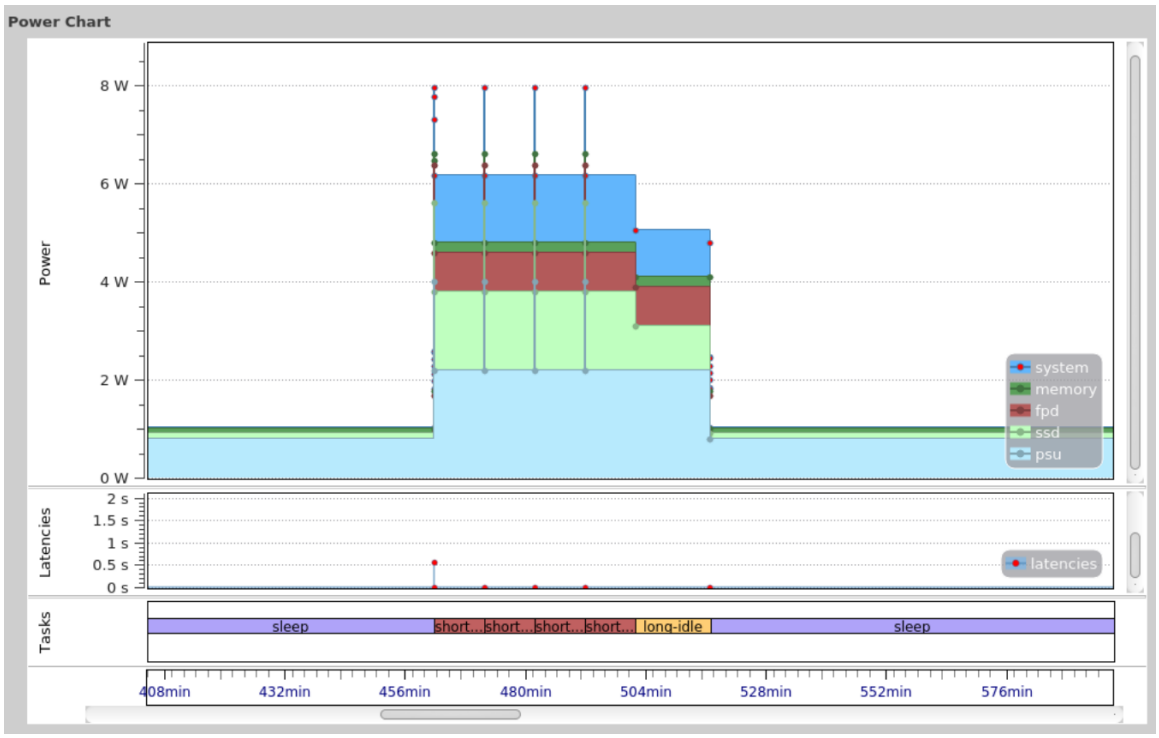
Source: Aggios, Inc.

The latency graph reflects the actual latencies of the Zynq MPSoC platform, where the boot time has been kept at 60 seconds to be conservative.

The simulation also shows the distribution of the power consumption, with the majority being taken up by the PSU, the SSD drive, as well as the FPD, which represents the “full-power domain.” The FPD includes the ARM-based quadcore-CPU of the Zynq MPSoC.

When looking at one of the activity cycles in detail, one can see the latencies for the transitions from sleep to active.

Figure 23: An Enlargement of an Activity Cycle Based on Sporadic User Activity



Source: Aggios, Inc.

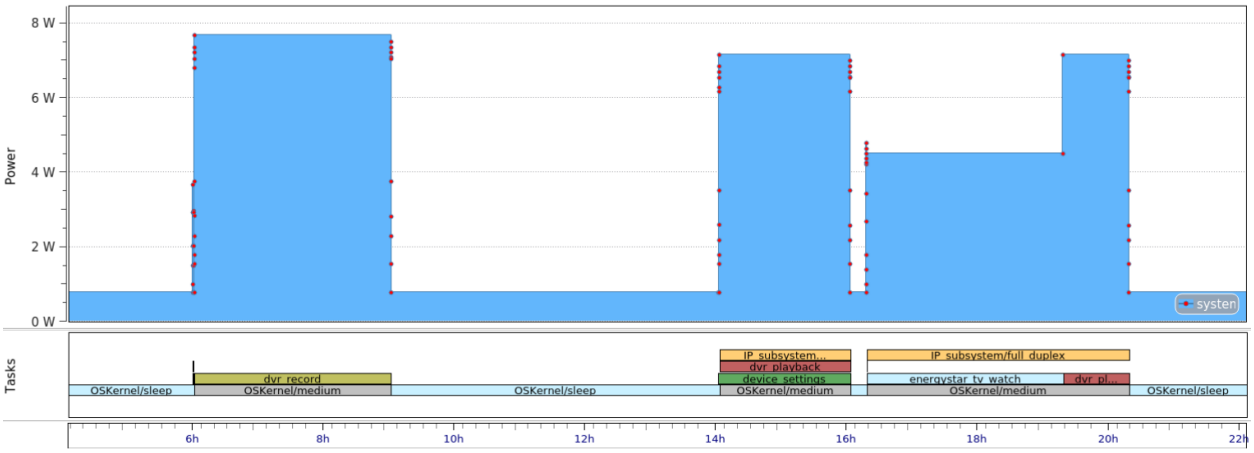
In **Error! Reference source not found.**, one can also see the transitions from the sleep state and the entering of short-idle and long-idle states, interrupt by occasional user activity.

Set-Top Box Virtual Prototype

Simulations of the virtual prototype of the set-top box was performed using EnergyLab Version 0.2.001. The simulation covers a detailed virtual prototype based on the Xilinx Zynq Ultrascale+ MPSoC platform.

The set-top box implemented contains all features that a typical set-top box has but uses only minimum amount of power. The features being demonstrated here include DVR record/playback and TV watch. The power consumptions of these tasks are presented in **Error! Reference source not found.**.

Figure 24: Feature Demos of the Set-Top Box Virtual Prototype



Source: Aggios, Inc.

Gaming Console Virtual Prototype

The research team performed simulations of the virtual prototype of the gaming console using EnergyLab Version 0.2.001. The simulation covers both the high-level example model of a gaming console using idealized components, as well as a detailed virtual prototype based on the Xilinx Zynq Ultrascale+ MPSoC platform.

Power Consumption Under Different Operating Modes

The results in Figure 25 show an improvement in all the primary modes simulated (in addition to other less-used modes). A minor improvement is observed in the game play mode (not the primary focus of this work), but more important are the results in streaming, navigation, and standby modes. Particularly the power improvement achieved in standby mode deserves consideration. The power usage has been decreased by a factor of six, from 8.8 W on average to 1.5 W.

Figure 25: Power Consumption Comparison Between the Gaming Console Virtual Prototype and a Typical Gaming Console

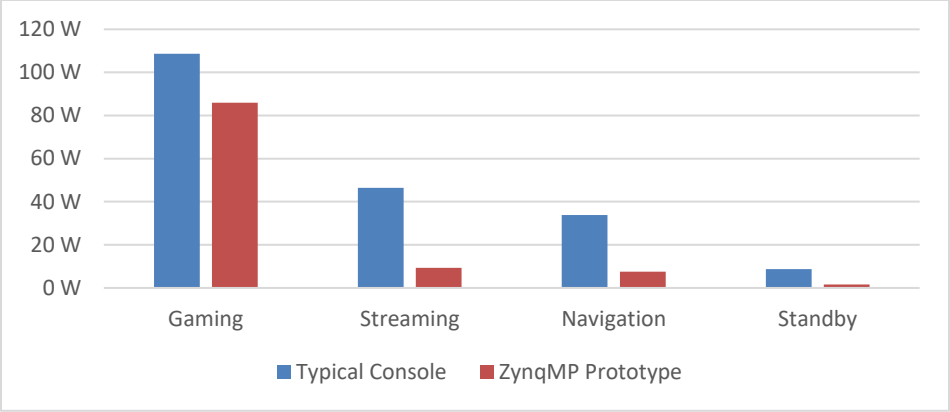


Table 10 shows expected power savings, comparing the typical console design with the virtual prototype. The bulk of the savings occurs in the standby mode; the amount of time spent here, combined with the large increase in efficiency of the prototype, creates the greatest energy savings. Substantial efficiency improvements have been achieved in the streaming state. These results highlight the fact that future power efficiency improvements should focus on streaming and standby modes.

Table 10: Expected Power Savings From the Gaming Console Virtual Prototype

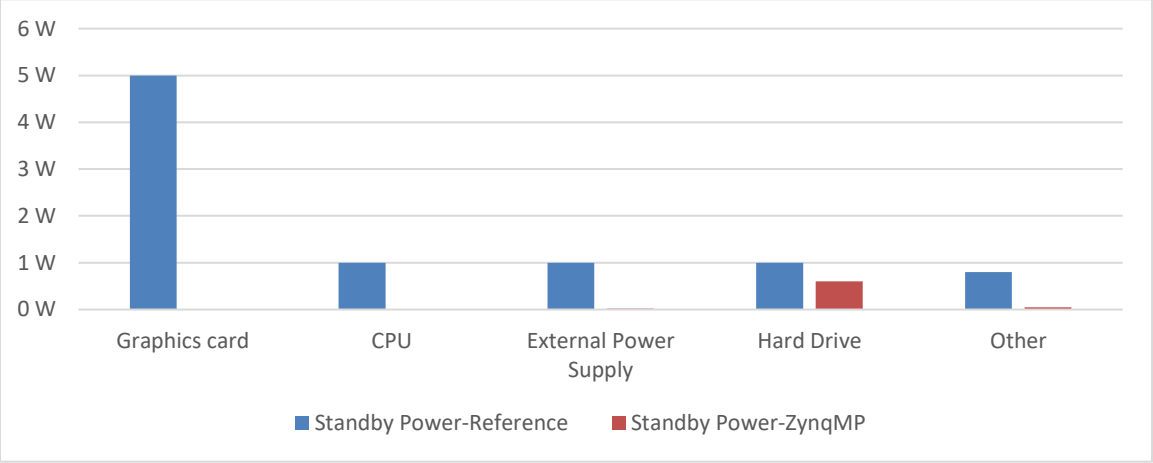
State	Typical Console	Virtual Prototype	Power Savings	h/day	Energy Savings/yr
Off	0.0 W	0.0 W	0 W	12.5 h	0.0 kWh
Standby	8.8 W	1.6 W	7.2 W	9.6 h	25.2 kWh
Navigation	33.8 W	7.5 W	26.3 W	0.1 h	1.0 kWh
Streaming	46.4 W	9.3 W	37.1 W	0.7 h	9.5 kWh
Gaming	108.6 W	85.9 W	22.7 W	1.1 h	9.1 kWh
				Total	44.8 kWh

Source: Aggios, Inc.

The table above shows the relative power improvements of the virtual prototype under each operating mode. From these results, the primary power improvements are achieved by reduction of graphics card power in noncompute-heavy modes. The full results of component usage in different modes are available through Energylab simulation.

Error! Reference source not found. demonstrates the power consumption breakdowns when the gaming console is in standby mode. While it has been demonstrated that most major components have the potential for significant power savings, the majority of the power savings comes from the fact that the graphics card used in the traditional gaming console is unable to cut power consumption to a level close to zero in the standby mode.

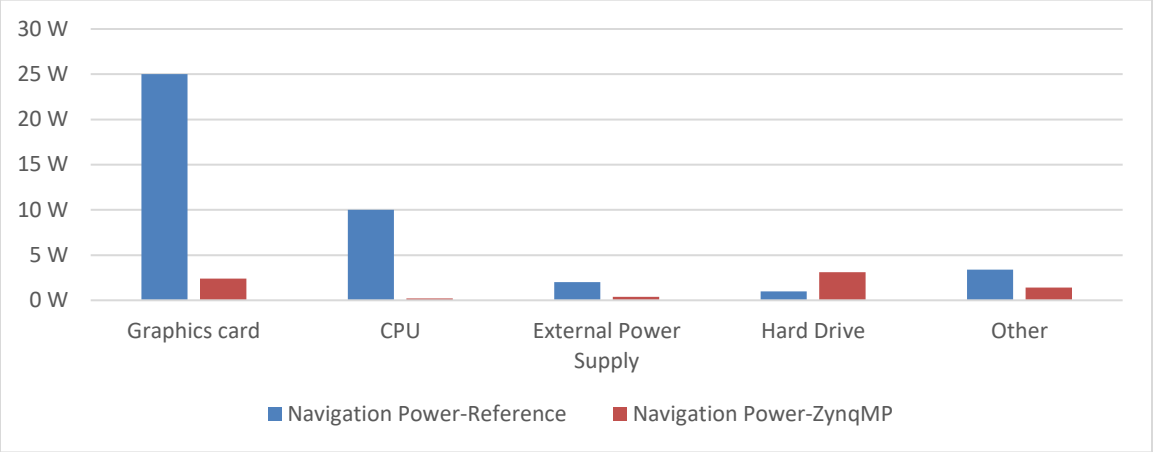
Figure 26: Per Component Power Savings of the Gaming Console Virtual Prototype in Standby Mode



Source: Aggios, Inc.

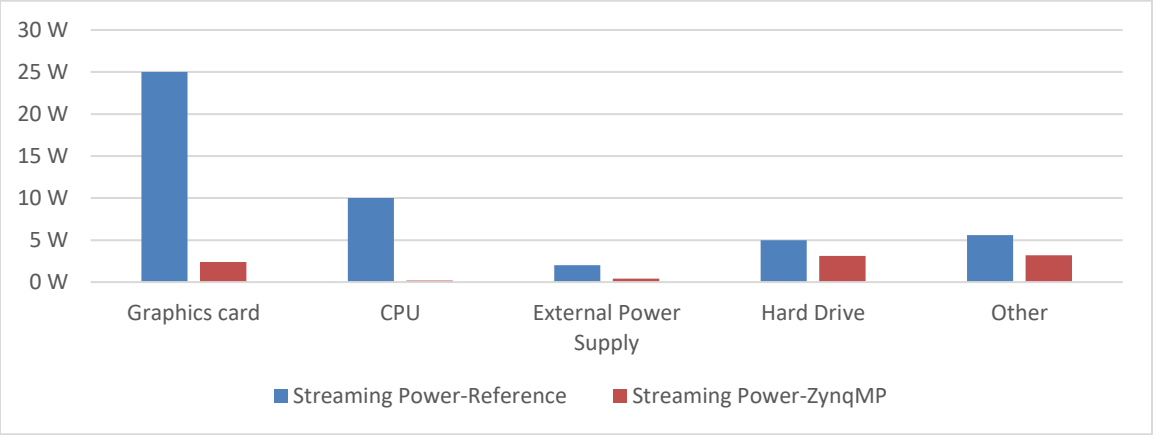
Further, **Error! Reference source not found.** demonstrates the power consumption comparison of the virtual prototype and the traditional gaming console. In most cases internet navigation is usually not graphics-intensive. Traditional gaming consoles demonstrate a huge inefficiency in power use under this operating mode. As many mobile devices are used for navigation as well, it is feasible in technologies to achieve small power consumption when performing navigation tasks. Similarly, **Error! Reference source not found.** demonstrates the per-component power comparisons under the streaming mode.

Figure 27: Per Component Power Savings of the Gaming Console Virtual Prototype in Navigation Mode



Source: Aggios, Inc.

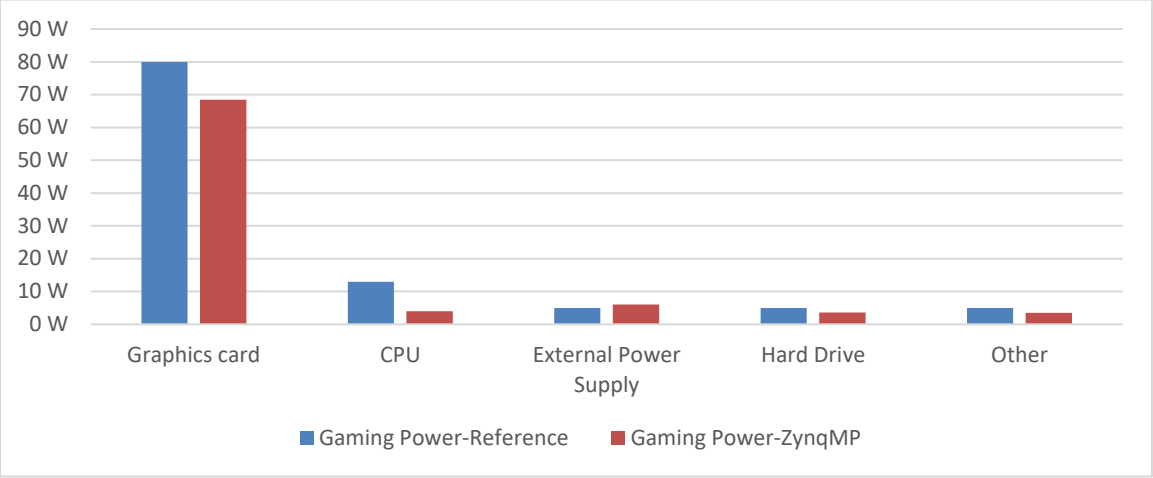
Figure 28: Per Component Power Savings of the Gaming Console Virtual Prototype in Streaming Mode



Source: Aggios, Inc.

When it comes to the gaming mode, the power savings of the virtual prototype is less significant, as demonstrated in **Error! Reference source not found.**. In this mode, most components are running at capacity, and the majority part of the power depends on the performance the component delivers.

Figure 29: Per Component Power Savings of the Gaming Console Virtual Prototype in Gaming Mode



Source: Aggios, Inc.

Power Analysis in Multiday Usage Scenarios

In the last section, the power savings of the virtual prototype under different operating modes have been presented. In reality, a gaming console does not stay in one mode forever. A typical user only uses the gaming console for a limited amount of time per day.

The next section aims to show how the overall power consumption of the virtual prototype compares with a traditional gaming console under a multiday use trace. The usage statistics collected have been summarized in **Error! Reference source not found.**.

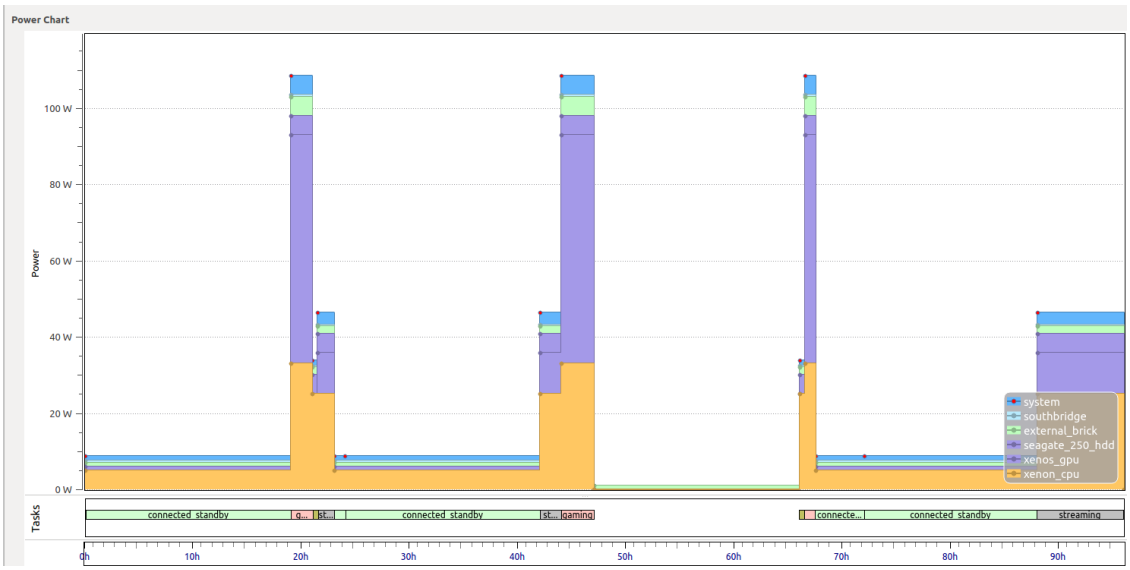
Tradition Gaming Console Model

The high-level prototype of the gaming console represents an example gaming console representative of the seventh generation of systems. This particular console model was based on Microsoft's Xbox 360.

The scenario used represents a four-day period, featuring the console in all available modes.

The usage profile below assumes that over four days, the system is being used to game on three of the four days for periods ranging from one to three hours. The system is largely on standby, although it is turned off after the second day of use. The user navigates the system for 30 minutes on two days, and the user streams video for one hour, three hours, and eight hours on the first, second, and fourth days.

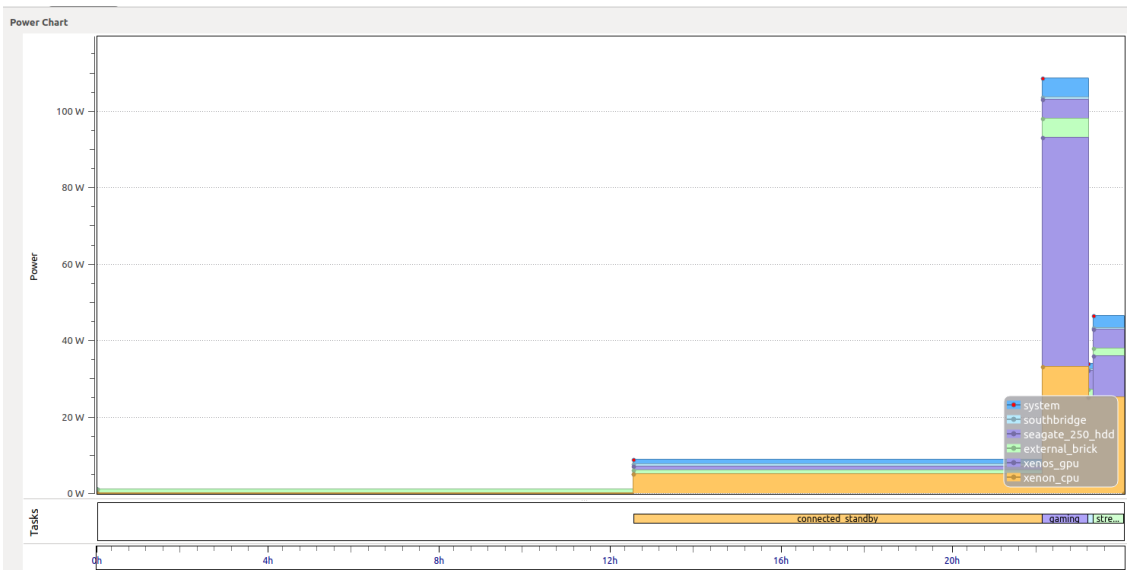
Figure 30: Power and Latency Graph for a Streaming-Heavy 96-Hour Cycle of the Reference Gaming Console Model



Source: Aggios, Inc.

The majority of system power is consumed in gaming modes, although long periods of streaming also cause high power use. Even with the inflated amounts of streaming used in this scenario, one still sees a large amount of power consumed in standby modes.

Figure 31: Power and Latency Graph for a Typical 24-Hour Cycle of the Reference Gaming Console Model

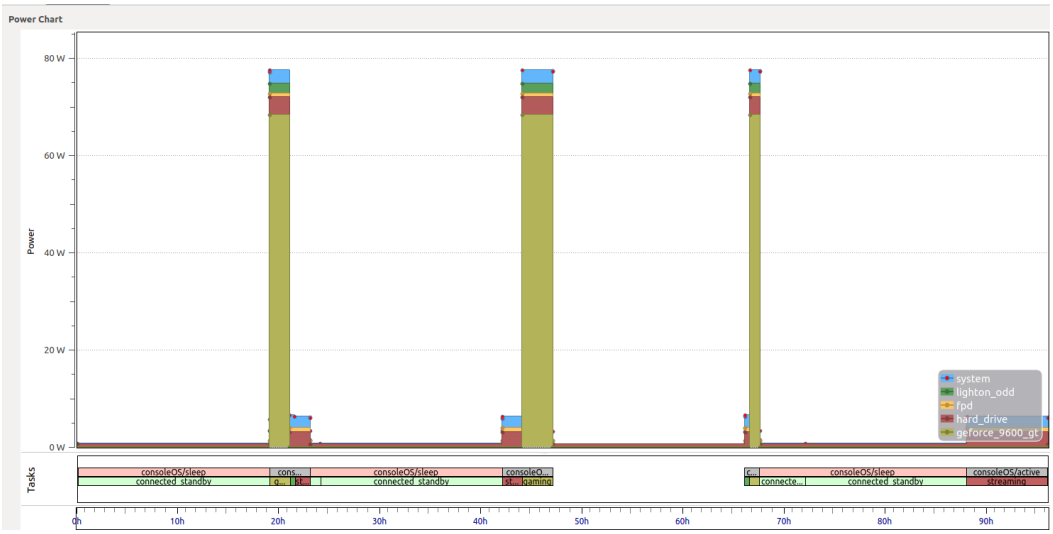


Source: Aggios, Inc.

The Gaming Console Virtual Prototype

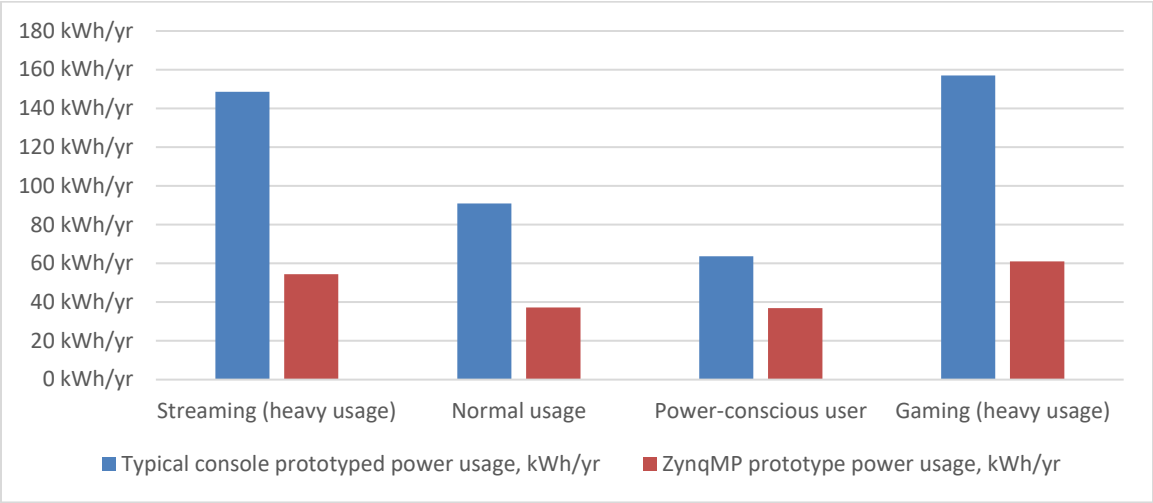
The same stimulus used to drive the previous model was also applied to the virtual prototype. This model reflects the power consumption of an energy-efficient ARM-based platform; hence, the power consumption is significantly lower than in the model based on current systems.

Figure 32: Power and Latency Graph for a Streaming-Heavy 96-Hour Cycle of the Zynq MPSoC-Based Model



Source: Aggios, Inc.

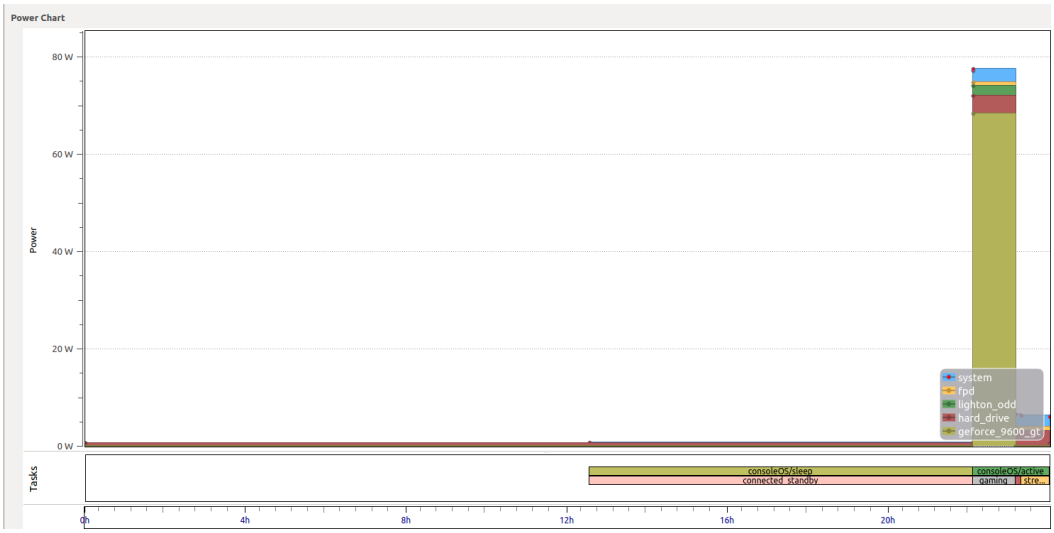
Figure 33: Yearly Power Usage of the Gaming Console Virtual Prototype



Source: Aggios, Inc.

The simulation also shows the distribution of the power consumption, with the vast majority being taken by the external graphics card in gaming modes. The card also consumes power while passing through streaming or navigational content to the television, but the bulk of the power is again consumed in gaming modes. Researchers also see large advantages in terms of power consumed in standby; the ability of the Zynq MPSoC to enter low-power states and resume allows the research team to achieve much greater power efficiency.

Figure 34: Power and Latency Graph for a Normal 24-Hour Cycle of the Zynq MPSoC-Based Model



Source: Aggios, Inc.

Discussions

The research team's example shows extreme promise in power savings for the gaming console. With an off-the-shelf, powerful graphics chip, one can still achieve great power savings. Furthermore, incorporating a graphics card into the chip itself, without the additional unnecessary overhead of the existing graphics chip (ARM Mali-400 MP2) embedded in the SoC,

offers promise for reducing power. Overall, the team's testing shows great promise for energy-efficient methods.

Table 11: Power Savings of the Gaming Console Virtual Prototype

Scenario	Zynq102 prototype power usage, kWh/yr	Gaming console prototyped power usage, kWh/yr
Streaming (heavy usage)	54.5	148.6
Normal usage	37.2	91
Power-conscious user	36.9	63.7
Gaming (heavy usage)	61	157

Source: Aggios, Inc.

Using More Energy-Efficient Components in Gaming Consoles

As expected, most of the power consumed in the highest-intensity mode (Game Play) was consumed by the graphics card. This is to be expected—as gaming consoles continue to increase in power, more of this increase is taken by the graphics card. Microsoft's recently released Xbox One X, for example, has a graphics card 4.6 times more powerful than the predecessor (the Xbox One), while the CPU of the console is only 30 percent faster (Boyle & Leger, 2018). The reasons for this discrepancy are outside the scope of this project but point to a primary focus on graphics cards when analyzing power consumption in the future.

Unfortunately, the results in this virtual prototype cannot provide substantial evidence about graphics card capability due to the design principle that virtual prototypes shall avoid simulating functionalities. The less extensive graphical capabilities of the Zynq102 forced the research team to turn to an external graphics card, with a correspondingly smaller power efficiency. Graphics card development should be examined in other studies. The authors, however, see the advantage of allowing graphical pass-through by examining graphics card usage in nongaming states.

The CPU shows more promising improvement. Streaming and navigation take very little CPU power, and using a low-powered CPU core shows huge power improvements in these areas. (Comparisons can be made to stand-alone navigation/streaming devices, such as the Apple TV or Roku, which take 2.4 or 3.5 watts respectively (Cunningham, 2015).) Moreover, as noted above, CPU power has become an increasingly minor console selling point as further efforts are directed at graphics cards.

Extensively Using Low-Power States

As mentioned in the above section, the gaming state drew heavy amounts of console power. While this virtual prototype was not well-suited for exploring power advantages in this area, the authors' prototype did draw correspondingly lower amounts in simulation and on a component-by-component level showed high amounts of improvement.

While the state with the highest total power draw in both the reference and prototype consoles was the gaming state, the standby state also drew large amounts of power due to the amount of time spent in the state. Due to the default settings of the Xbox (Delforge & Horowitz, 2014), the console enters the more power-intensive standby state with instant on functionality instead of the slightly higher-latency off modes.

Standby showed great potential of the use of power management controllers, as shown by the example of the Zynq MPSoC. The ability to enter a low-power mode that can be resumed in less than a second has major implications for gaming consoles, which spend large amounts of time unused. Further research should be made in this area.

The research also reveals large improvements in the streaming and navigation areas. While such inefficiencies in current console designs have been noted by other studies, confirmatory examples from prototypes made with current components only emphasize the power inefficiencies of the existing console model.

An important note when comparing the research team's current prototype to future models is the existence of multiple components for the same purpose—for example, the dual external graphics card and the internal ARM Mali-400 MP2. While these components were cost-effective to use in the authors' prototypes, mass hardware production might make using duplicate components prohibitive. As time goes on and costs decrease, however, such duplication becomes more feasible.

Implications for Future Consoles

This research provides important information for future console developers. In particular, it points to the importance of ensuring low power draw in standby modes. Current systems show issues with system component dependency; the PlayStation 4, for example, takes 5 watts in standby mode to power USB ports (Ars Technica Addendum , 2015). A power-focused system design would ensure that such components necessary in standby mode should be capable of being powered on independent of high-power-draw components. The research team's reference design shows that such results are achievable.

The research team's results also show the importance of managing the streaming power draw. Current console systems perform extremely inefficiently when streaming, often using an order of magnitude more power than devices designed specifically for streaming. The team's reference design uses a smaller amount of power, making use of the bypass mode of the graphics card. As streaming becomes more common for gaming consoles, power efficiency in streaming modes becomes correspondingly more important.

The research team's research and prototypes show the utility of dedicated power management units and the use of power-focused simulation and prototyping. This research will hopefully be useful in future game console planning and development.

CHAPTER 4: Reference Designs

The research team developed reference designs for each of the plug-load devices analyzed for the project. These designs were built with energy efficiency as a primary goal while aiming to achieve functional capabilities in line with current market devices.

Reference designs were built after creating virtual prototypes for each device, following the process outlined in the methodology section. Each design used a similar base for simplicity of development and to enable reuse of core components. Component and architecture choice and a discussion of the results follow.

Components Implementation

Hardware Platform

The research team used the following principles as the basis for selecting the project component:

1. The reference design shall contain hardware components that provide functionalities of the target platform.
2. The hardware implementation shall adopt silicon designs for the recent mobile technologies used on high-end cellphones or tablets.
3. The hardware implementation shall be measurable in terms of latency and power consumption for evaluations and demonstrations.
4. If it would be cost-prohibitive to implement some necessary components in the hardware, the components shall be simulated with the power consumption and state transition latencies accounted for.
5. The reference design shall implement basic software functionalities to demonstrate the power and latency consequences when running the software.

Based on the above principles, the research team opted to use the Xilinx ZCU100 board as the platform for the plug-load device reference designs. The research team used Xilinx ZCU102 board as a measurement testbed, outlined below in the testbed section. The ZCU100 board is small and relatively power-efficient, allowing the research team to demonstrate the power consumption of an efficient device.

Components

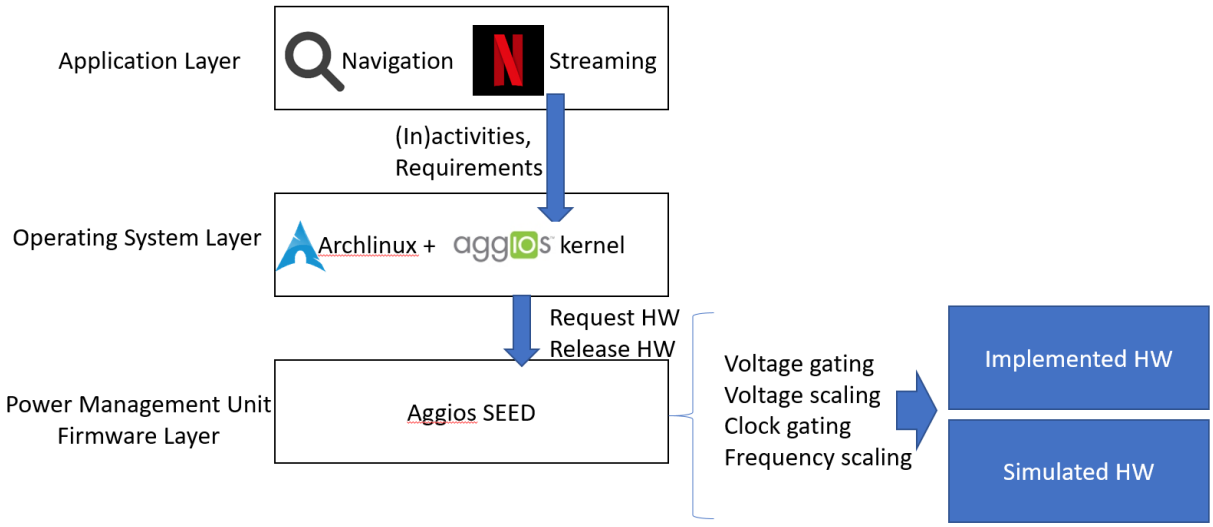
All plug-load devices have a similar base, requiring CPUs, system memory, networking, and graphical processing. The ZCU100 possesses all these components; when necessary, the more powerful components that may be required to achieve functional parity with current market devices are incorporated into the power analysis. The ZCU100 also contains a field-

programmable gate array (FPGA), which can be programmed to function as additional components. In addition to the base components, the following components are implemented in FPGA or simply included in the power analysis:

- Set-top box: SATA SSD, cablecard, cryptographic accelerator
- Console: discrete graphics card, SATA SSD, additional processor
- Computer: internal power supply (IPS), SATA SSD, discrete graphics card
- Television: power supply, TV screen, LED backlight

Operating System and Application Software

Figure 35: Architecture Diagram of the Set-Top Box Reference Design



Source: Aggios, Inc

The reference design consists of several layers. The above three-layer application model was used in all designs, with user-facing software running on top of ArchLinux+, which performs power management actions through AGGIOS SEED running at the PMU firmware layer. A brief overview of each layer follows.

Application Layer

This layer includes the user-facing software used to navigate, stream videos, and run the user interface for the reference design. This information is passed to the operating system layer for further power management actions.

This design is typical of mobile applications. For example, Android provides APIs for the application developers to inform the OS that it needs or no longer needs a certain hardware. For many significant hardware components, this is performed via the wake lock APIs.

Operating System Layer

The operating system layer implements a modified version of Archlinux, which is a popular Linux distribution. The Linux kernel used is a modified version that allows the collection of requests from the application layer, aggregation, and the relaying of the requests to the power management unit firmware layer. This layer also implements a debug interface as a means to inspect the internal settings of the power management unit firmware.

Power Management Unit Firmware Layer

The power management unit (PMU) is a set of firmware that runs on a dedicated processor. This processor is a low-power MicroBlaze processor with a maximum power consumption of only 4 milliwatts (mW). Because it is dedicated to power management, the logic in the firmware will never be preempted by the potentially busy software running on the main cores, i.e., the application processing units. The processor running the PMU firmware is invisible to the operating system; the OS cannot schedule tasks on it. The OS communicates with the PMU firmware through a set of APIs that are expected to be standardized through this project.

The design of the PMU is an improvement over the existing mobile technologies. Today, designers of mobile operating systems choose to use software components to perform similar tasks. The hardware solution chosen has unique advantages in that it can perform management tasks while the main processors are sleeping, allowing more power-saving potential.

Tasks that the PMU firmware performs include:

- Collecting of request/release commands from the operating system.
- Using the fewest number of hardware components and lowest possible states to fulfill the requirements of the reference design when the platform is active.
- Putting the hardware components into the lowest states when the system is inactive
- Adjusting the clock frequencies of the core components such that the lowest possible frequency is used while meeting the performance goals.
- Adjusting the voltage levels of the core components such that the lowest possible voltage level is used while meeting the performance goals.

Testbed

The team used two systems to measure power. Initially, reference designs were tested for AC power consumption. As the project progressed, final numbers were achieved using a DC power measurement system. A variety of software and hardware tools were used to measure results. The project also included testing of the reference designs, including functional verification of the ability of each device to perform tasks appropriate to the plug-load functions implemented.

Hardware Tools

AC Measurement

Initial power consumption tests measured device AC power consumption. The research team measured the AC power drawn by the device power supply; this is the measurement that most indicates the device power draw on the actual power grid. In all cases, the power supply shipped with the device was used. This method gave good initial estimates.

However, the team's primary concern was not device draw on the power grid, but a relative comparison of devices, including current devices and the physical prototype. The power wasted in conversion by the power supply was not relevant to this device comparison. Therefore, for increased precision and a more accurate view of device power used, the research team developed DC measurement techniques.

The team performed initial AC measurements using the Yokogawa WT310. The device measured the power drawn from an attached power strip and sent the data to an attached measurement computer, where the WTViewer software was used for initial measurements. These data were then output to Excel. The WT310 provided adequate sampling rate and log accessibility for the initial phase of the project.

DC Measurement

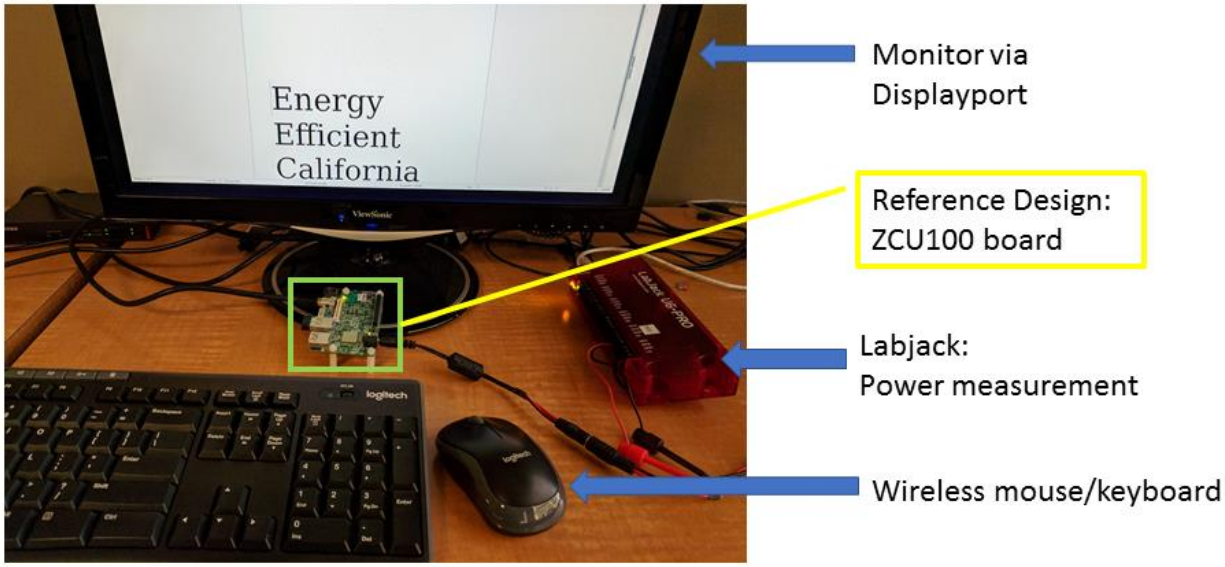
Initial efforts to perform device measurement used AC, but DC measurement provided a more accurate view of the power the device consumed, as opposed to the amount lost in conversion. Using DC measurement also provides more accurate moment-by-moment views of device power; the conversion performed by the power supply gave the team less meaningful point-by-point data. Accordingly, the research switched to using DC measurement to measure the total power of the ZCU100 board.

The primary physical measurement device used for DC power was the LabJack U6-PRO. The power supply cables of the ZCU100 were modified to add a 100 mOhm resistor (Ohmite 12FR100E) through one wire. Clamps connecting either side of the resistor to the AN0 and AN1 LabJack terminals were then added. In this way, DC power consumed was measured before accounting for the conversion loss of the power supply.

This resistor does introduce a very small increase in the power consumed; a small portion of the power is consumed heating the resistor. However, the 12 V difference implies low amperage and a corresponding low drop. A resistor with low Ohm resistance was intentionally chosen to account for this fact.

See the following example of the computer reference design being tested. The design is performing a word processing task, using a wireless mouse and keyboard and with power consumption being measured through the LabJack U6.

Figure 36: Reference Design Testing Set-up



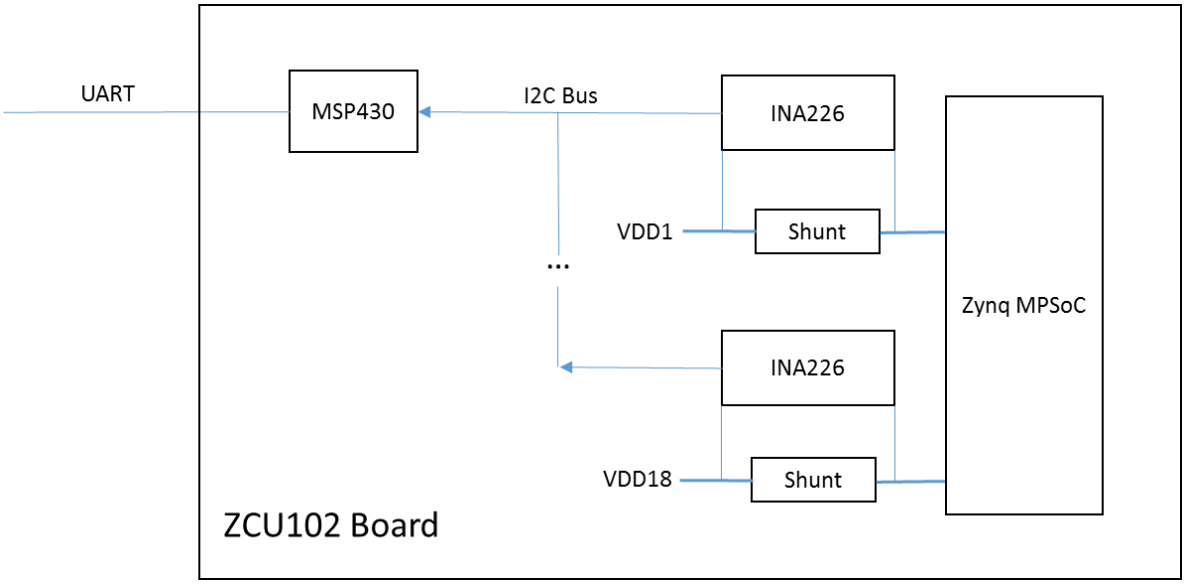
Source: Aggios, Inc

Software Tools for per-Rail Power Measurements

In addition to measuring the DC power, the research team had to measure particular active components that were drawing power. To this end, measurements were made on the ZCU102 board. This board features voltage rail measurement devices, measurement of the power consumption of specific components, and better determination of the sources of power consumption.

For power rail measurement, AGGIOS uses the on-board power measurement capabilities of the ZCU102: the MSP430 microcontroller. The MSP430 is a mixed-signal microcontroller (including analog and digital circuits) designed for low-cost and low-power applications. To measure and output data from the MSP430, previously developed Energylab functionality is used. The MSP430 reads data from several INA226 current monitors, each attached to a voltage rail measuring the differential on either side of a shunt resistor. These data are read over the I^2C bus, necessitating disabling Archlinux access to the bus to avoid collisions. With this method, more detailed information about the power domains and particular systems responsible for power draw is obtainable.

Figure 37: Measurement Diagram of ZCU102 Rail Measurement System



Source: Aggios, Inc

Test Plan

The research team designed the test plans with the following considerations:

- Physical power measures are preferred over simulations whenever possible. Because the simulation results are delivered mainly by the prototype developed for this project, the focus on the reference design should be on the measurements from the hardware.
- DC measurements are preferred over AC measurements, if possible. Because the computers components operate directly on DC power, measuring DC gives a more precise insight on power usage. In comparison, measuring on the AC power will incur inaccuracies if the efficiency of the power supply is not a constant number.
- Latency measurements shall be performed via software on the board itself. While it is possible to measure the latency via external devices for a lower overhead added to the runtime system, a software measurement performed by taking time stamps from the real-time clock (RTC) inside the board is more accurate. This is because the software has more precise knowledge on the moment when the component switches to the desired states.

Television Test Plan

These tests are designed to verify the ability of the reference design to perform common television tasks with lower power draw. These tests also verify that the reference design implements important smart television functionality.

TV Tuning

As TV tuning requires an important hardware component (digital TV tuner), which is simulated only in the reference design, the TV tuning functional test cannot be performed by receiving actual signals and playing them. Instead, AGGIOS performed the testing using a hybrid approach. The approach used was as follows:

- For TV components that are implemented in simulation, which include the digital TV tuner, the TV screen, and the backlight, the tester shall verify that the components are turned on at the right time and all the respective dependencies are satisfied in the simulation.
- For TV components that are implemented on the hardware platform, i.e., the ZCU100 board, the tester shall verify that the software runs with no bugs and the hardware components that are required, either by the simulated hardware or by the implemented hardware, are in the correct states.

Video Streaming

Video streaming testing shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The experiments shall include streaming video files from online sources with a variety of resolutions.

Implementation of the reference design contains software with a UI that allows the user to select videos from a catalog and stream them. The content of the videos shall from online sources and are streamed to the device via the Wi-Fi connection. The ability of the prototype to run YouTube and Netflix apps directly shall also be verified.

Real-time power consumption shall be measured via DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply.

USB Gallery

This test shall be conducted using the TV reference software. This software shall be able to:

- Detect the event that a USB device has been plugged into.
- Find photos in the USB drive.
- Display the photos in full screen.
- Switch to the next photo on a predefined interval.

The test shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort™.

Real-time power consumption shall be measured via DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including the losses in the internal power supply and the simulated screen.

Computer Test Plan

These tests are designed to verify the ability of the reference design to perform common computer tasks with lower power consumption. These tests also verify that the reference design fully implements all important computer functionalities.

Web Browsing

The Web browsing test shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The test shall include two major Web browsers: Chromium and Mozilla Firefox.

Real-time power consumption shall be measured via DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and in the SATA SSD.

Word Processing

The Microsoft Word processing test shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The tests shall include Libre Office.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and the SATA SSD.

Video Streaming

The video streaming tests shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The tests shall include streaming video files from online sources with a variety of resolutions.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and the SATA SSD.

Set-Top Box Test Plan

These tests are designed to test the power consumption of the reference design while performing conventional tasks. The measurements conducted on the reference design reveal lower power consumption than seen in conventional set-top box designs.

Sleep Mode

Sleep mode testing shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and the SATA SSD.

Navigation Mode

Navigation mode testing shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and the SATA SSD.

Video Streaming

Video streaming testing shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The experiments shall include streaming video files from online sources with a variety of resolutions. This experiment represents the primary functionality of the set-top box reference design.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the internal power supply and the SATA SSD.

Gaming Console Test Plan

These tests are designed to verify that the reference design can achieve lower power usage in common use scenarios. The power requirements of the reference design have been compared to the power requirements of a traditional console. The team also verified that the reference design developed for this project is able to provide adequate performance on these tests.

Navigation

The navigation tests shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The experiment shall use an HTML5-based navigation system running on either a Chromium or Firefox browser.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the discrete graphics card and the SATA SSD.

Video Streaming

The video streaming tests shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The experiments shall include streaming video files from common online sources with a variety of resolutions.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the discrete graphics card and the SATA SSD.

Gaming

Gaming tests shall be conducted on a ZCU100 board, with an external monitor connected via DisplayPort. The experiments shall include running freeware games on the reference design to verify performance.

Real-time power consumption shall be measured via the DC power supplies. The measured power numbers shall be adjusted to account for the simulated devices, including losses in the discrete graphics card and the SATA SSD.

This gaming testing does not incorporate high-performance gaming. Due to the lack of a PCIE slot, the reference design is not capable of including a high-performing graphics card, which is the most important component for playing complex modern games. The central processor of the reference design is also substantially less powerful than processors included in actual consoles, providing a barrier to performing high-performance testing.

This report attempts to address this issue by focusing on testing navigation and streaming, two important use cases for current gaming consoles. These tests further simulate high-powered components to gain a better perspective on what a power-efficient system should look like. This approach should work well for analysis of potential future power-efficient gaming consoles.

Measurements and Power-Saving Analysis

Latency Results

Power management is meaningless without reasonable latencies. If latency is not a concern, devices can simply power off to enter the lowest possible power state and boot only when necessary. From a functional perspective, however, this scenario is unacceptable to users. This section demonstrates that power-efficient hardware and power management choices have small latency costs and what hardware costs do exist are dwarfed by software latency, which can be fixed through judicious choice of software.

Measurement Method

The research team measured latency using custom PMU Firmware. The research team developed a library to track latency between particular code points and report on the results. The original data collection methods used debug prints, but measured latency increased with these debug prints enabled. The completed solution tracked results internally, only outputting data when the latency costs could be safely ignored. Particular code points were chosen representing particular components entering the on state; these were used to measure the latency of each component. The specific code points chosen are outside the scope of this report, but the research team believes the functions chosen represent the real-time costs of resumption for the hardware devices chosen.

This library uses a simple data structure to note the time stamp of each entry, which is stored internally. The PMU clock was used to note the times. Further confirmation of the accuracy of these times was obtained by intentionally implementing large latencies and comparing the results reported by these libraries to the wall clock time; in all cases, the AGGIOS library proved accurate. Below is an example output of the latency measurement printout. These prints occur after system resume to avoid adding additional latency.

Figure 38: Latency Measurement Printouts

```
[   41.498878] Restarting tasks ... done.
FPD latency: 10.4 ms
DDR latency: 12.1 ms
APU latency: 1.0 ms
Total HW latency: 23.4 ms
root@xilinx-zcu102-2017_4:~#
```

Source: Aggios, Inc

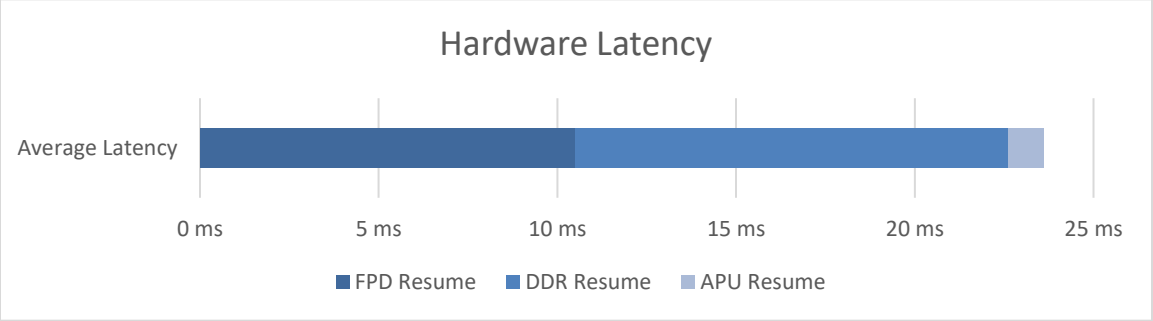
For software latency, the research team took base numbers for early and device resume directly from Linux reports. The latency for userspace return was obtained through a combination of reported Linux latency and code points. A call through the Linux PMU API was added to indicate control returning to user space. The difference between the final hardware code point and control returning to user space was taken and the other sources of software latency were subtracted, giving a result. The PMU API call adds a small amount of latency to the measured numbers, but this amount should be negligible in comparison to the total user space latency.

Measurement Results

The inclusion of latency measurement numbers in this report has two purposes. First, the inclusion demonstrates that the latency of the transition between low-power and active states is sufficiently low as to make the low-powered state a reasonable choice for users. This is shown through latency measurements demonstrating the time from a wakeup interrupt being received to the time control returns to Linux user space applications. The reference designs show an average resume time of 330 ms, roughly a third of a second. This latency is sufficiently low as to make the low-power state not costly to users.

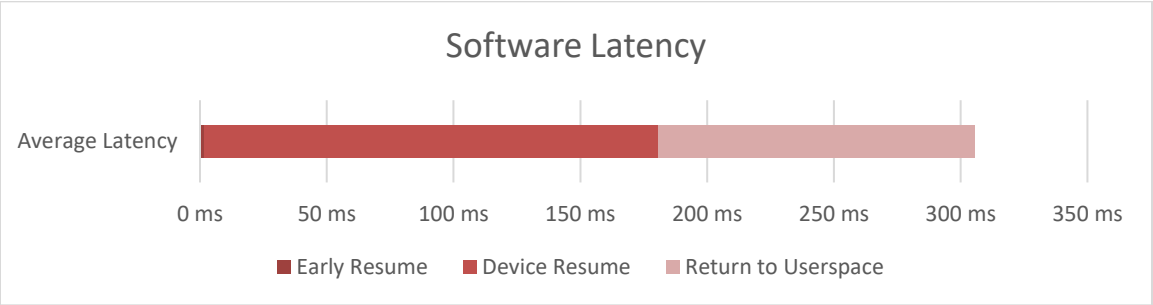
The second purpose is to demonstrate the relative costs of hardware and software latency. This purpose demonstrates earlier claims that software latency is the most significant factor in resuming from suspension, with hardware latency costs only a minor portion of total latency. The hardware latencies averaged 23.6 milliseconds, more than 10 times lower than the average software latency of 306 milliseconds. While these numbers are sensitive to the choice of board and software used, this provides substantial evidence toward the role of software latency in device state transitions.

Figure 39: Hardware Latency Breakdown by Component: Full-Power Domain, DDR SDRAM, and Application Processing Unit



Source: Aggios, Inc

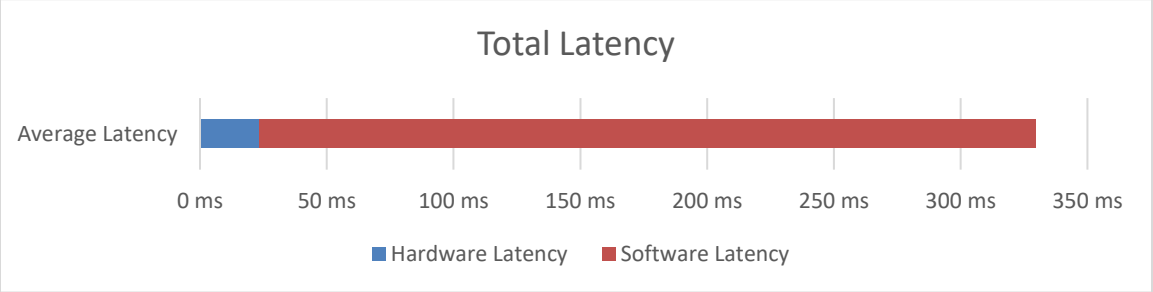
Figure 40: Software Latency Breakdown Based on Logged Linux information



Early resume takes less than 2 milliseconds.

Source: Aggios, Inc

Figure 41: Total Latency Breakdown Into Hardware and Software Latency



Source: Aggios, Inc

Challenges

The largest issue the research team encountered while trying to perform device latency measurements was caused by a bug in the PMU ROM code. During device suspend/resume, the programmable logic domain was disabled to save power. This triggered a previously unreported

bug that caused a timeout, making a single supply check call take 65 milliseconds—a substantial fraction of the total suspend/resume time.

Further investigation showed that the bug was caused by faulty logic. The ROM code, while powering up the FPD, performed a check to determine if the programmable logic domain was enabled. The function performed this determination, but it additionally introduced a large overhead when the programmable logic domain was off. For the research team's latency measurements, the research team performed a workaround to obtain better results. A fix for the bug will be included in future Xilinx releases, although the fix may result in slightly higher latency than the research team's workaround. This fix should not affect these results, however; as noted above, hardware latency is dwarfed by software latency.

TV Reference Design Power Analysis

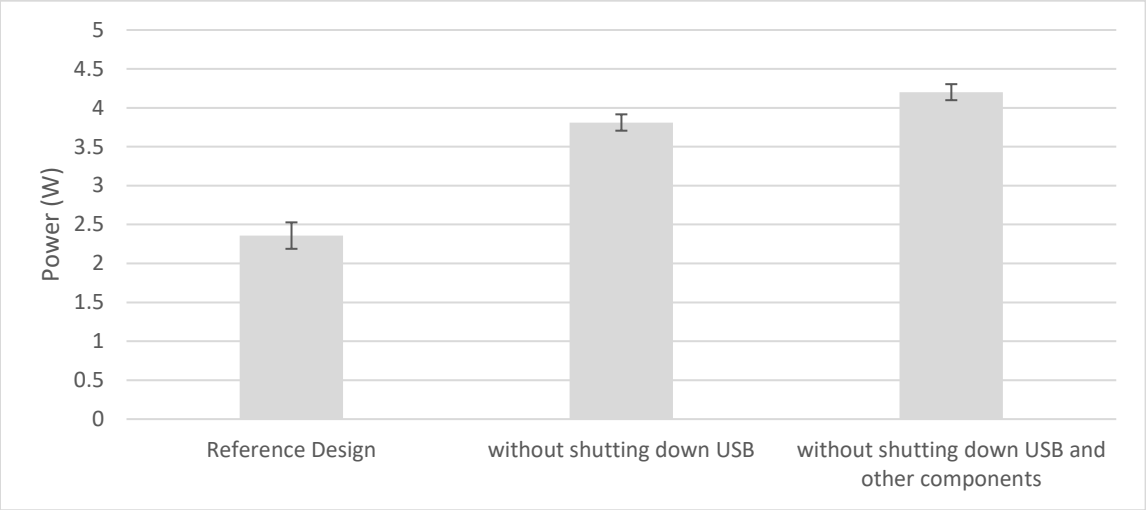
For the TV reference design, fewer improvements are shown than in other reference designs. Television power consumption is dominated by the choice of screen, which is not a focus of this project. However, even though analog power consumption dominates digital power consumption in television hardware, the reference design still demonstrates improvements in the digital realm when performing the USB gallery tasks. Even in television hardware, power-efficient designs demonstrate improvements in power efficiency. This improvement in the USB gallery task demonstrates the mobile design principle of minimum hardware for task requirements.

Minimum Active Hardware for Task Requirements

The implementation of the USB gallery tasks is described as follows.

- When the USB gallery task is selected from the main menu of the TV, the USB device is immediately mounted to the filesystem.
- Unnecessary components are immediately put into the lowest possible power state. Examples of these components include the networking components, the FPGA (PL) part that contains the logic that powers the digital TV tuner, and so forth.
- The content of the USB storage device is scanned. Some photos are selected and copied to the internal storage area of the device, which is a microSD card in this implementation.
- The external USB storage device is unmounted.
- Power to the USB controllers and the external USB device is cut off.

Figure 42: USB Gallery: Power Consumption With Standard Deviation



Source: Aggios, Inc

The power consumption of the reference design is plotted in **Error! Reference source not found.** and compared with two baselines.

The first baseline is the same as the television reference design but without shutting down the USB controller and the power to the external USB device. This baseline demonstrates 1.523 W or 39.2% of savings. These savings are because the USB controller on the board and the external USB storage device consume significant amounts of power. Many external USB storage devices are not designed for the optimal power efficiency and pull an unnecessary amount of current. While televisions manufacturers have little control on the power efficiencies of the external USB storage devices, these experiments show that it is feasible for televisions to shut down external devices for power savings.

The second baseline is the same as the reference design but without shutting down any unused component at all. This baseline represents a typical TV design that lacks the carefully implemented power management solutions and keeps every component inefficiently powered on at all times. Compared with this baseline, the reference design consumes 1.843 W or 43.9% less power.

Computer Reference Design Power Analysis

The research team compared the reference design to a baseline of a traditional computer. The baseline computer chosen is a Dell Latitude 5570 laptop. The reason for this choice of baseline computer is that, as laptops face battery life requirements, the manufacturers are motivated to optimize the power consumption of the design. Therefore, unlike other high-performance computers such as Web servers, laptops contain state-of-the-art power optimization designs. Using a laptop as a baseline is more likely to provide a reasonable comparison against the reference design.

When comparing with the baseline, the power consumption impacts of screens were forestalled by disabling the built-in screen of the laptop. The display output of the laptop was connected to the same monitor used to connect to the reference design.

During the experiment, the battery of the laptop was plugged in for sufficient time to ensure that the battery is always 100% full. Though a plugged battery consumes a small amount of additional power, this configuration was used as a baseline because it represents the typical use case for a laptop in California homes.

The default power options set by the manufacturer (Dell) were used, which is the balanced power plan. This setting dims the display after 2 minutes of inactivity, turns off the display after 5 minutes of inactivity, and puts the computer to sleep mode after 30 minutes of inactivity.

Purely based on measured results, the authors obtain the following results for the total energy efficiency of the reference design when compared to existing hardware. The authors focus on the frequently occupied computer states where most of the power is consumed and see the following improvements in power consumption.

Table 12: Power Consumption and Savings of Computer Reference Design

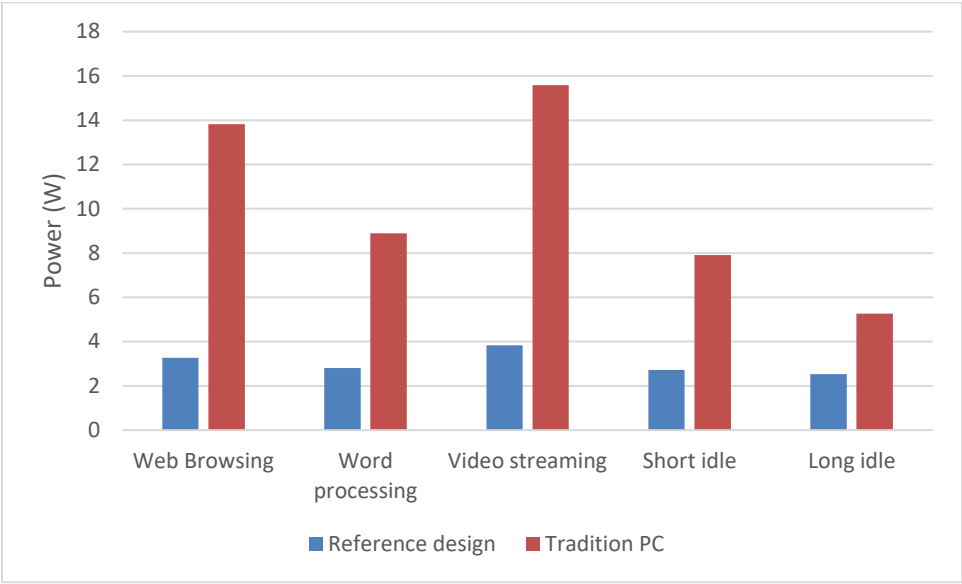
Application	Traditional PC	Reference Design	Power Savings	Hours/Day	Energy Savings/Year
Short Idle	7.91 W	2.713 W	5.197 W	7.2 hr	13 kWh
Long Idle	5.269 W	2.525 W	2.744 W	1.2 hr	1.2 kWh
Total					14.2 kWh

Source: Aggios, Inc

Offloading Tasks to Dedicated Hardware

These results demonstrate that a device with the ability to offload particular noncomputationally expensive tasks onto dedicated hardware like the ZCU100 can demonstrate substantial hardware savings. This method is far more power-efficient than using high-powered hardware for simple word processing and video streaming. The diagram below demonstrates the results of running three noncomputationally intensive tasks on the reference design and on a traditional computer, providing short and long idle power consumption as comparison points.

Figure 43: Power Consumption of the Reference Design Compared With a Traditional PC



Source: Aggios, Inc

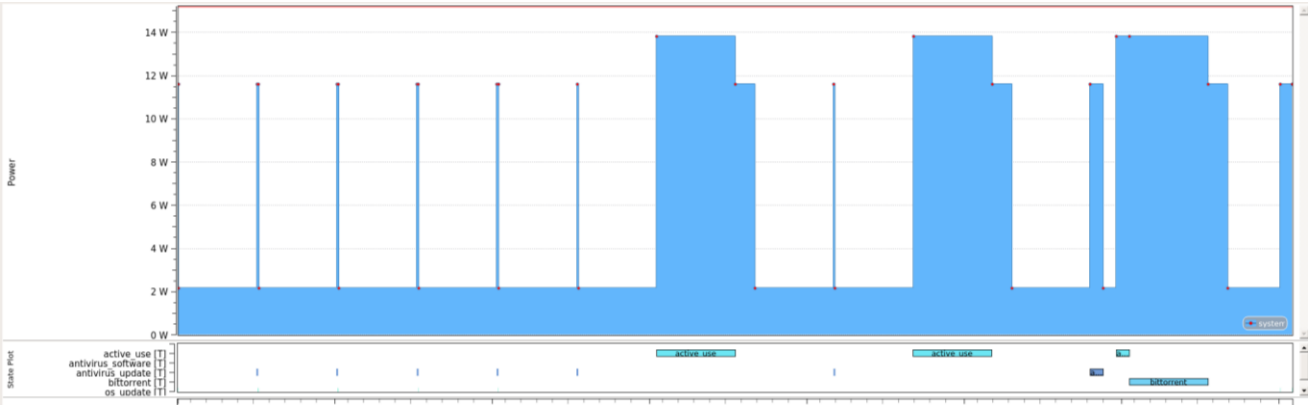
The results demonstrate that, compared with a traditional PC, which is already optimized for power efficiency, offloading to dedicated hardware results in significant power savings. The percentage of power savings depends highly on the power consumption of the baseline computer, which is typically less power-efficient than the dedicated hardware solution.

White-Listing Applications

Android-powered mobile devices use a whitelist to allow only a subset of tasks to keep the device awake. In this subsection, the potential power benefits of using a similar technology in PCs are demonstrated.

The assumed usage pattern is plotted in **Error! Reference source not found.**. In the 14 hours of run time, the computer is used for two periods with one hour each. The rest of the time contains an hour of BitTorrent downloading, and periodical runs of OS update, antivirus software update, antivirus software. The computer is assumed to have a 15-minute timeout before going to sleep again when it is activated.

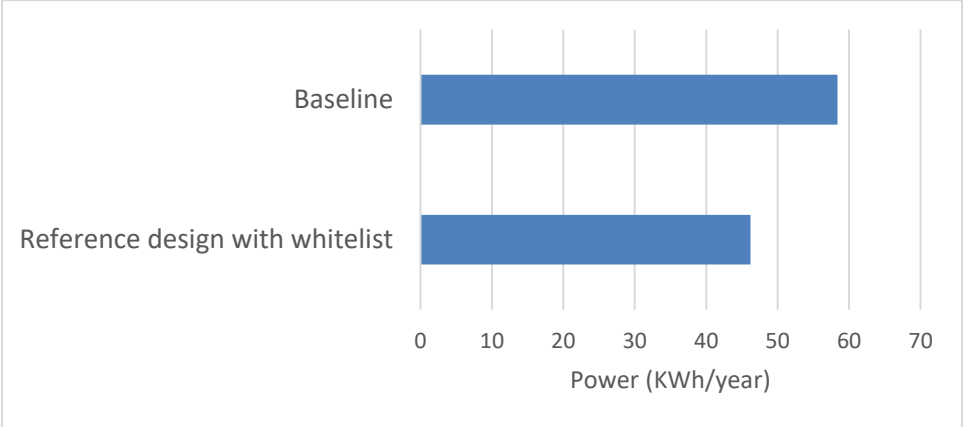
Figure 44: Assumed Usage Pattern in 14 Hours



Source: Aggios, Inc

This experiment of the power management algorithm features two whitelisted tasks: BitTorrent and antivirus software. This is because these tasks, though typically run in the background, are generally expected to be kept active for as long as the user desires. In contrast, other tasks, despite invocation of OS APIs to prevent the system from entering sleep states, are not expected by the user to keep the system on.

Figure 45: Power Comparison Average to a Year



Source: Aggios, Inc

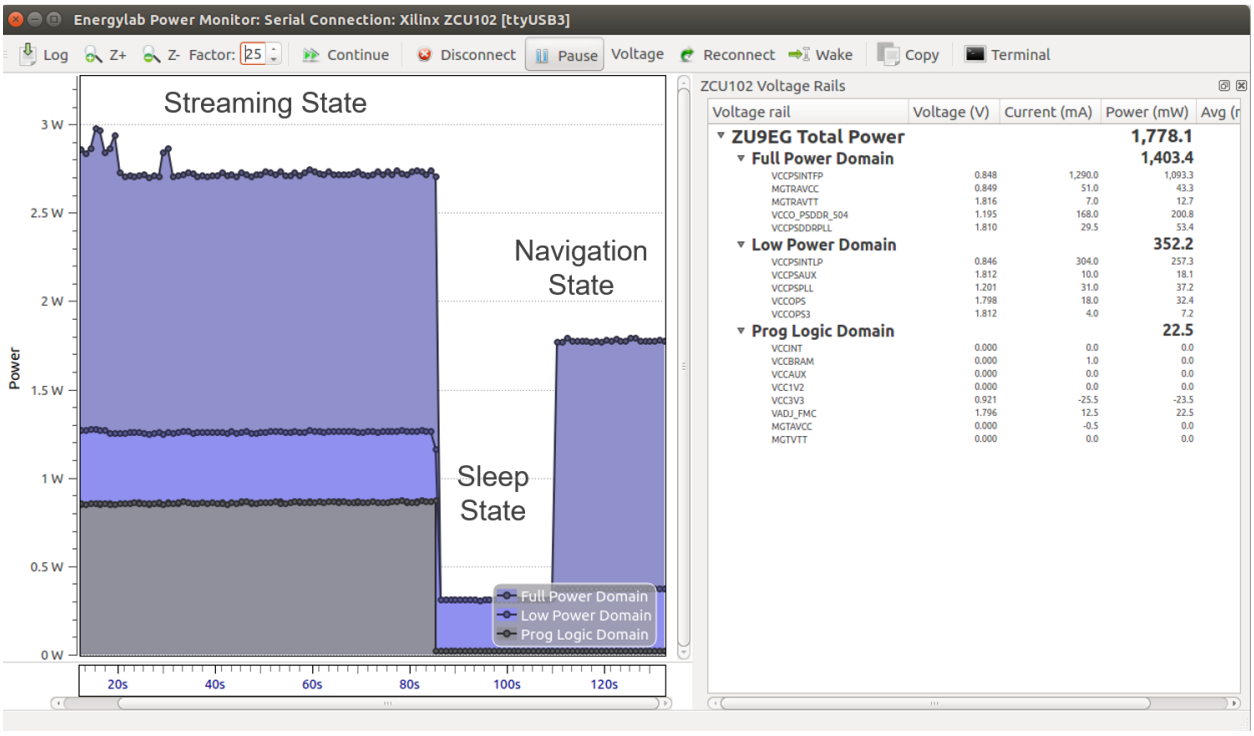
The results plotted in **Error! Reference source not found.** show a power consumption of 46.2 kWh/year compared with 58.4 kWh/year from the baseline. These power savings are partially because nonwhitelisted tasks can no longer keep the system awake. Additional power savings can be attributed to the long idle time the non-hitelisted tasks no longer cause; in a traditional design, nonessential tasks not only cause wake time themselves, but force a computer to wait an additional length of time before reentering sleep.

Set-Top Box Reference Design Power Analysis

The research team compared the results of the reference design to conventional set-top boxes as measured in the NRDC's "Improving the Efficiency of Set-Top Boxes." Active mode

navigation, active mode streaming, and sleep power consumption are compared to those of conventional designs. Below is an example of the system power consumption in streaming, sleep, and navigation states. The results are captured through EnergyLab's power monitor functionality. The programmable logic domain (in gray) is inactive and consumes no power in the final state; this finding demonstrates power improvements obtainable through the disabling of unnecessary devices.

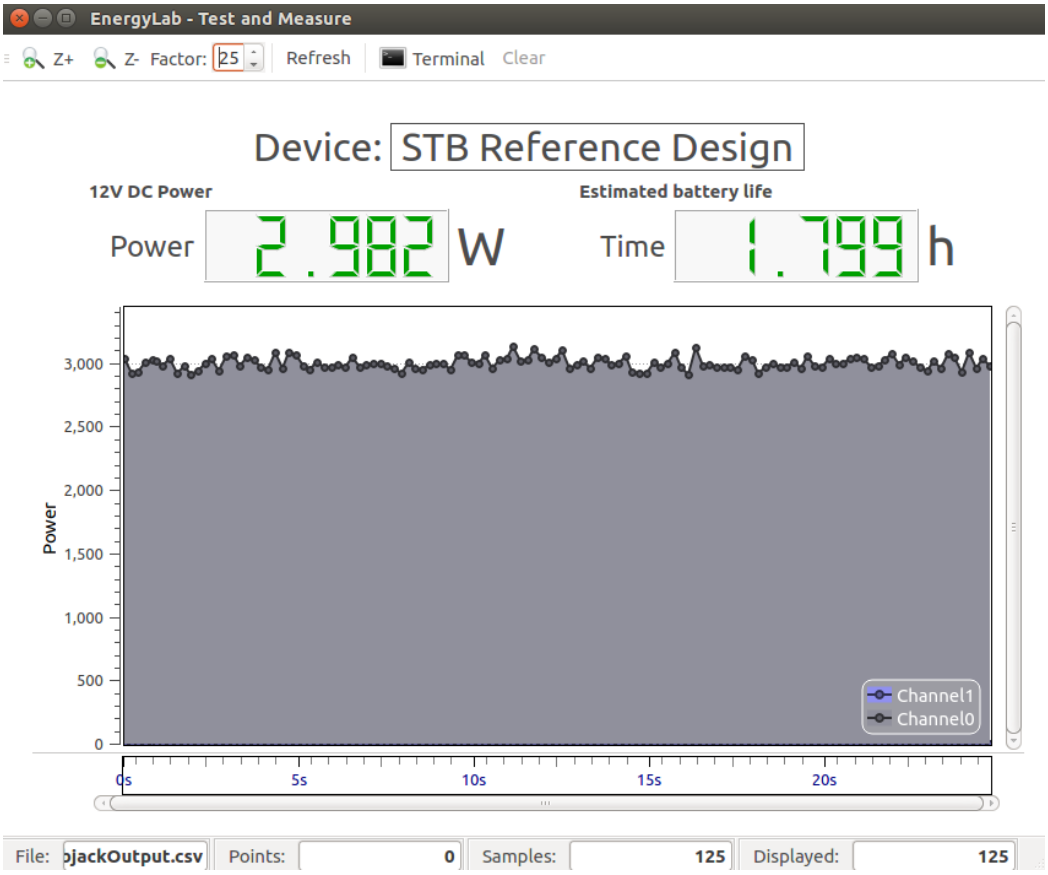
Figure 46: Set-Top Box Reference Design Power Measurements



Source: Aggios, Inc

See also the following picture of DC measurement results, again captured using EnergyLab. This picture demonstrates device power consumption in active mode.

Figure 47: DC Power Measurement of ZCU100



Source: Aggios, Inc

While this design is not capable of achieving power as low as dedicated streaming devices due to the simulated hardware used, it shows substantial power savings compared to current devices. Even with conservative assumptions regarding power draw in inactive mode, assuming the reference design still requires full power for the CableCard device in sleep mode, large power savings can be achieved. See Table 13 for the breakdown of different components contributing to the power consumption numbers.

Table 13: Breakdown of Power Consumption by Component

	Active State	Navigation State	Idle State
ZynqMP Chip	2.7 W	2.3 W	0.2 W
Other ZCU100 Board	0.6 W	0.6 W	1.3 W
SATA Drive	0.4 W	0.4 W	0.0 W
CableCard (CableLabs, 2015)	3.3 W	3.3 W	3.3 W

Cryptographic Accelerator (Singh & Kumar, 2012)	1.3 W	0.0 W	0.0 W
Total Power	8.2 W	6.5 W	4.6 W

Source: Aggios, Inc

The ZynqMP chip numbers are obtained from observations of the ZynqMP chip behavior on the ZCU102 board. The other board power numbers were obtained through observing actual ZCU100 behavior and factoring out chip power consumption.

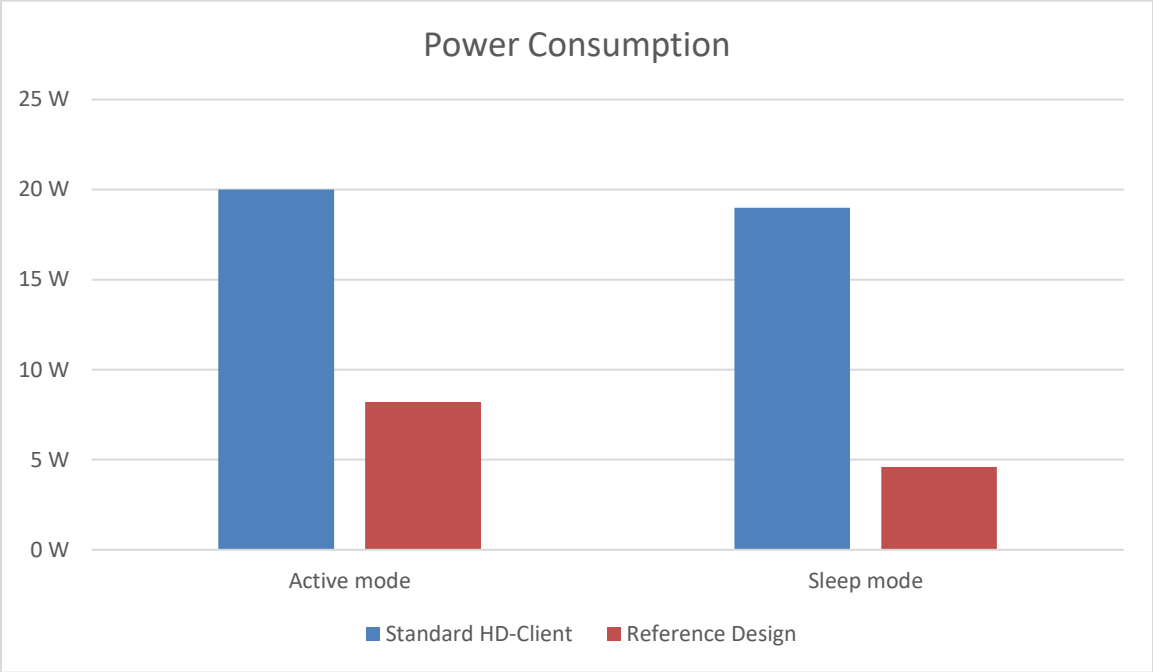
Other power numbers are for simulated components. The states that require each component should be self-explanatory. The SATA drive is required when actively viewing and recording streams or navigating through the same but is unnecessary in idle states. The cryptographic accelerator is necessary only in active modes. CableCard is modeled as on in all states to maintain communication with upstream providers. The authors use hours/day numbers from the NRDC's report and conservatively assume the majority of set-top box use time is spent streaming video content rather than in in navigation states. With these assumptions, the lower power consumption numbers shown in Table 14 are obtained.

Table 14: Set-Top Box Power and Energy Savings

Use Case	Standard HD-Client	Reference Design	Power Savings	Hours/day	Energy Savings/year
Streaming	20 W	8.2 W	11.8 W	3 hr	12.9 kWh
Navigation	20 W	6.5 W	13.5 W	0 hr	0 kWh
Sleep	19 W	4.6 W	14.4 W	21 hr	110 kWh
Total					122 kWh

Source: Aggios, Inc

Figure 48: Comparison of Reference Design and the Baseline



Source: Aggios, Inc

Gaming Console Reference Design Power Analysis

Power results are compared to an established baseline: a traditional console. Microsoft's Xbox One was chosen as the baseline console. The Xbox One is a representative eighth-generation console and provides a fair representation of the current console market.

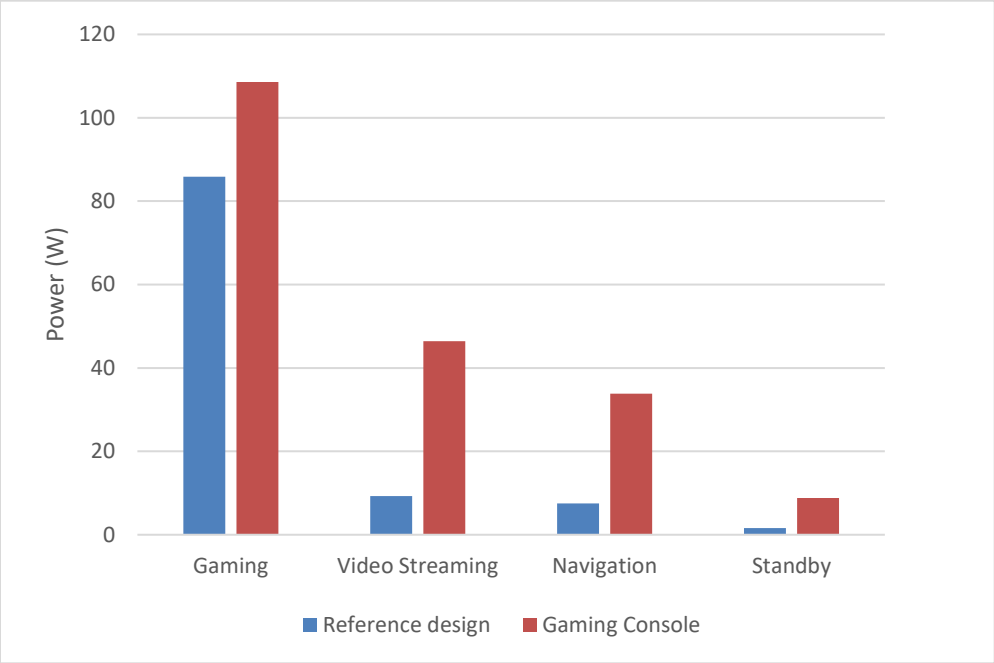
Xbox One numbers were sourced from the National Resource Defense Council's 2014 report. Its testing procedure mimicked the guidelines from U.S. EPA's Energy-Efficient Game Console recognition program; for further details, the EPA documentation may be consulted. The same procedures were used in the reference design measurement. Power consumption of the reference design and the power consumption of the baseline implementations information were collected in the Table 15. Hours/day information on the usage of consoles was sourced from https://www.sciencedirect.com/science/article/pii/S0301421513003923.

Table 15: Gaming ConsolePower and Energy Savings

Use Case	Xbox One	Reference Design	Power Savings	Hours/day	Energy Savings/year
Gaming	108.6 W	85.9 W	22.7 W	1.1 hr	9.1 kWh
Streaming	46.4 W	9.3 W	37.1 W	0.7 hr	9.5 kWh
Navigation	33.8 W	7.5 W	26.3 W	0.1 hr	1.0 kWh
Standby	8.8 W	1.6 W	7.2 W	9.6 hr	25.2 kWh
Total					44.8 kWh

Source: Aggios, Inc

Figure 49: Comparison of Reference Design to Traditional Console



Source: Aggios, Inc

These results demonstrate that this heterogeneous architecture has resulted in significant power savings compared to current gaming consoles. The focus of this project is on low-performance modes, and high power savings are shown in streaming, navigation, and standby modes. These results are not unexpected; other writers have noted (Delforge & Horowitz, 2014) unnecessarily high power consumption of the traditional console for these tasks. The Apple TV and Google Chromecast are capable of streaming with less than 2 watts of power; it is unsurprising that a design optimized for power efficiency can perform the same task using fewer than 10 watts.

Even if the low-power architecture discussed is not scalable to high-performance requirements, this solution can be used to improve power efficiency. The low-performance, low-power design

may be used as a separate cluster in navigation and streaming modes, while a higher-performance cluster is used while gaming. Such an architecture would have high latency and upfront manufacturing costs but would be extremely power-efficient.

Summary of Reference Designs

The results presented in this section demonstrate the value of a power-efficient design approach. Mobile design approaches like forcing users to explicitly whitelist applications can show major improvements in power efficiency. Using low-power hardware for low-performance tasks can achieve major power efficiency improvements over present designs. These set-top box and gaming console prototypes show major improvements when performing common tasks. Furthermore, all these results can be obtained without meaningful losses in user-facing latency. Power-efficient design provides a powerful approach to decreasing the energy consumption of plug-load devices.

CHAPTER 5: Conclusions

In this project, the research team applied energy-efficient design principles to the development of four common plug-load devices to demonstrate the applicability of those design principles to mobile devices.

The work done in the first half of the project has shown how the energy-first method can be applied in the early phases of a project by modeling the energy behavior of devices in virtual prototypes using the UHA description format. The work has also demonstrated how these models estimate and optimize the energy efficiency of such devices by keeping components in the lowest possible power states that support the application requirements.

Through the creation of reference designs for each targeted device, the project has also demonstrated how the mobile efficiency principles can help design more energy-efficient devices using the results obtained from the virtual prototyping phase to define and optimize the power states of the physical devices. This helped produce reference designs that achieved significant improvements in energy efficiency for each targeted device, where the ultimate savings ranged from 2.4 kWh/year for smart TVs up to 70 kWh/year for set-top boxes.

Applying the energy-first development method proposed in this project to the four target devices has helped improve and extend the method and the UHA description format. This application also identified best practices for modeling different types of devices at different abstraction levels, including the energy-related details of various components, including the related dependencies and state-dependent power models to achieve results that match the actual behavior of the real hardware. At the same time the knowledge gained

The standardization efforts on the system-level energy-modeling front have led to a proposed IEEE standard that is ready for balloting in the IEEE P2415 working group with expected completion by 2020. The efforts promote the proliferation and application of the energy efficient design method.

Further Work and Next Steps

While this project successfully demonstrated the applicability of energy-efficient design principles for plug-load devices, the benefits to California will only materialize if the industry adopts this method and embraces energy efficiency for mainstream electronic devices found in California households.

An appropriate next step would be to launch a collaborative project with one or more of the industry players to produce an actual device following these design principles, proving to the rest of the community the commercial viability of this approach.

CHAPTER 6: Technology Transfer

One objective of this project was to make the developed technology available to the industry. This objective has been pursued via two avenues: standardization and proliferation.

Standardization

One part of the technology transfer effort was to standardized the unified hardware abstraction and the EEMI interface through the IEEE standardization body. For this purpose, the IEEE P2415 project has been started (Reference: https://standards.ieee.org/project/2415.html).

The status of the standardization is that a final proposal has been submitted to the P2415 working group. The standard has the opportunity to be adopted in the next one to two years.

Proliferation

To promote the developed technology in front of the industry, the findings from this project have been presented at several forums.

On February 8, 2018, the research was invited by the National Resources Defense Council to present the findings from the mobile efficiency project to the set-top box community during their set-top box voluntary agreement meeting in Denver.

On August 15, 2018, the research team held an emerging technologies webinar hosted by the California Energy Commission to present the results of the EPIC project to a wide audience from the industry as well as regulators. The webinar had more than 140 attendees.

REFERENCES

Arm Limited. (2016). *Cortex-A53*. Retrieved from Arm Developer: https://developer.arm.com/products/processors/cortex-a/cortex-a53

ARM Limited. (2017). *ARM Power State Coordination Interface Platform Design Document .* ARM.

Ars Technica Addendum . (2015, 3 28). *How "standby" modes on game consoles suck up energy*. Retrieved from Ars Technica: https://arstechnica.com/gaming/2015/03/how-standby-modes-on-game-consoles-suck-up-energy/

Bianchini, E. P. (2004). Energy conservation techniques for disk array-based servers. *Proceedings of the 18th annual international conference on Supercomputing (ICS '04)* (pp. 68-78). New York, NY, USA: ACM.

Boyle, E., & Leger, H. S. (2018, 09 10). *Xbox One vs Xbox One S: should you make the upgrade?* Retrieved from TechRadar: https://www.techradar.com/news/xbox-one-x-vs-xbox-one-s-vs-xbox-one-should-you-make-the-upgrade

CableLabs. (2015). *OpenCable CableCARD Interface 2.0 Specification.* Retrieved from https://specification-search.cablelabs.com/opencable-cablecard-interface-2-0-specification/

CableLabs. (2017). *DOCSIS® 3.1 – A New Generation of Cable Technology*. Retrieved from CableLabs: https://www.cablelabs.com/specifications-library/docsis/

Cunningham, A. (2015, 11 4). *New Apple TV wants to be more than just a streaming box (but it isn't yet)*. Retrieved from Ars Technica: https://arstechnica.com/gadgets/2015/11/new-apple-tv-wants-to-be-more-than-just-a-streaming-box-but-it-isnt-yet/2/

Delforge, P., & Horowitz, N. (2014). The Latest-Generation Video Game Consoles: How Much Energy Do They Waste When You're Not Playing. *NRDC issue paper*.

Dong, M., Choi, Y.-S., & Zhong, L. (2009). Power Modeling of Graphical User Interfaces on OLED. *Proceedings of the 46th Annual Design Automation Conference.*

EnergyStar. (2014). *ENERGY STAR Program Requirements for Set-top Boxes.* Retrieved from https://www.energystar.gov/sites/default/files/specs//FINAL%20Version%204%201%20STB%20Program%20Requirements%20for%20Manufacturers.pdf

Linaro Limited. (2016, May 24). *Devicetree Specification Release 0.1.* Retrieved from Embedded Linux Wiki: https://www.devicetree.org/downloads/devicetree-specification-v0.1-20160524.pdf

Microchip. (2017). *USB5744: 4-Port SS/HS USB Controller Hub.* Retrieved from http://ww1.microchip.com/downloads/en/DeviceDoc/USB5744-Data-Sheet-DS00001855G.pdf

Micron. (2012). *Micron Serial NOR Flash Memory N25128A*. Retrieved from https://www.micron.com/resource-details/0a7815eb-4ee6-442d-9f63-61c7ddfda167

Microsoft. (2013, 3 13). *Xbox 360 manuals and specifications*. Retrieved from Xbox.com: https://support.xbox.com/en-US/xbox-360/console/manual-specs

Samsung Electronics. (2016). *65" Class KU7000 4K UHD TV - UN65KU7000FXZA*. Retrieved from Samsung US: https://www.samsung.com/us/televisions-home-theater/tvs/4k-uhd-tvs/65-class-ku7000-7-series-4k-uhd-tv-2016-model-un65ku7000fxza/

Singh, K. P., & Kumar, D. (2012). Performance evaluation of low power MIPS crypto processor based on cryptography algorithms. *International Journal of Engineering Research and Applications*, 1625-1635.

Ullah, I., & Whang, A. J.-W. (2015). Development of optical fiber-based daylighting system and its comparison. *Energies, 8*(7), 7185--7201.

United EFI Forum. (2017, 05). *Advanced Configuration and Power Interface Specification Version 6.2*. Retrieved from https://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf

VIZIO Inc. (2018). *VIZIO Tuner-Free Displays*. (VIZIO) Retrieved 6 1, 2017, from Vizio: https://www.vizio.com/tuner-free

Xilinx. (2017). *Embedded Energy Management Interface.* Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug1200-eemi-api.pdf

Xilinx. (2017). *ZCU102 Evaluation Board User Guide.* Xilinx.

GLOSSARY

Term	Definition
ABC	ambient brightness control
AC	alternating current
App	Computer program application or software
APU	Application Processing Unit
CPU	Central Processing Unit
DDR	double data rate
EEMI	embedded energy management interface
EPA	U.S. Environmental Protection Agency
GPU	Graphics Processing Unit.
IEEE	Institute of Electrical and Electronics Engineers
kHz	kilohertz
LCD	liquid crystal display
LED	light-emitting diode
MB	megabyte
ms	millisecond
mW	milliwatt
OCM0	On-chip memory bank 0
OLED	organic light-emitting diode
PCB	printed circuit board
PCIE	peripheral component interconnect express
PMU	Power Management Unit
RPU	Real-time Processing Unit
RTC	real-time clock
SoC	system on chip
TEC	typical energy consumption
TWh	Terawatt hour (one billion kilowatt-hours).

Term	Definition
UHA	Unified Hardware Abstraction
USB	universal serial bus
Wi-Fi	radio wireless local area networking
Xbox	video game console developed and owned by Microsoft

APPENDIX A: UHA Reference Manual

UHA Reference Manual

Unified Hardware Abstraction – Language Reference

Version 1.0

9/05/2017

Davorin Mista

Aggios® Inc.

Table of Contents

1.1 Scope 6
1.2 Purpose 6
1.3 Typographic conventions 6
2 Definitions, acronyms, and abbreviations 8
2.1 Definitions 8
2.2 Acronyms and abbreviations 8
3 Description principles 9
3.1 Programmer's view 9
3.2 Modularity and reuse 9
3.3 Description elements 9
4 Modeling basics 10
4.1 UHA language elements highlights 10
4.1.1 UHA objects 10
4.1.2 Control components 10
4.1.3 UHA properties 10
4.1.4 UHA parameters 11
4.2 Minimal UHA description 11
4.3 Managing larger descriptions 11
5 Elements 12
5.1 Object types 12
5.2 Object names 12
5.3 Properties 12
5.3.1 Data types for property values 13
5.3.2 Parameters 17
5.4 Object hierarchies 17
5.5 Abstract objects and inheritance 18
5.5.1 Defining abstract objects 18
5.5.2 Defining derived objects (directly) 19
5.5.3 Including objects as abstract 19
5.6 References to an object 19

6 Hardware components 21

6.1 Component definitions 21

6.2 Special properties 21

6.2.1 power 21

6.2.2 voltage-parent, clock-parent, and reset-parent 21

6.3 Capabilities 22

6.4 Power models of hierarchical components 22

7 States and transitions 24

7.1 States 24

7.2 State hierarchies 24

7.3 Capabilities of components and their states 25

7.3.1 Capability definitions 25

7.3.2 type 25

7.3.3 min and max 26

7.4 State dependencies 26

7.4.1 Asserting capability requirements 27

7.4.2 Requiring a shared capability 27

7.4.3 Requiring an additive capability 27

7.5 Special properties of states 27

7.5.1 set 28

7.5.2 power 28

7.5.3 entry-latency and exit-latency 28

7.5.4 default 28

7.6 Transitions 28

7.7 Special properties of transitions 29

7.7.1 from 29

7.7.2 latency 29

7.7.3 power 29

8 Control components 30

8.1 Modeling voltage and frequency 30

8.2 Declaring control objects 30

8.3 Clocks 30

8.3.1 States 31
8.3.2 Frequency-output capability 31
8.3.3 Modeling clock dividers 31
8.4 Resets 32
8.5 Voltage-supply 32
8.5.1 States 32
8.5.2 Voltage-output capability 32
8.5.3 Power-output capability 32
8.6 States driven by control components 33
8.6.1 Basic control component dependencies 33
8.6.2 Bidirectional control component dependencies 33
8.7 Dependencies on frequency and voltage 34
9 Events 35
10 Tasks and impacts 36
10.1 Tasks 36
10.2 Special properties for tasks 37
10.2.1 begin and end 37
10.2.2 duration 37
10.2.3 requires 37
10.2.4 set 37
10.2.5 power 37
10.3 Impacts 37
10.4 Special properties for impacts 38
10.4.1 begin and end 38
10.4.2 duration 38
10.4.3 requires 38
10.4.4 set 38
10.4.5 power 38
11 Power accounting 39
11.1 Power model selection 39
11.2 Power value calculation 39
11.3 Power value aggregation 39

11.4 Examples 39
12 UHA file structure **41**
Annex A (informative) **Bibliography** **42**

Overview

1.1 Scope

The new standard defines the syntax and semantics for energy oriented description of hardware, software and power management for electronic systems. It enables specifying, modeling, verifying, designing, managing, testing and measuring the energy features of the device, covering both the pre- and post-silicon design flow. On the hardware side the description covers enumeration of semiconductor intellectual property components (System on Chip, board, device), memory map, bus structure, interrupt logic, clock and reset tree, operating states and points, state transitions, energy and power attributes; on the software side the description covers software activities and events, scenarios, external influences (including user input) and operational constraints; and on the power management side the description covers activity dependent energy control. The new standard is compatible with the current and future IEEE 1801 (Unified Power Format (UPF)) standard to support an integrated design flow. It provides the higher level of abstraction and therefore enables earlier (more abstract) modeling of power states using UPF. Additionally, the new standard complements functional models in VHDL/Verilog/SystemVerilog/SystemC by providing an abstraction of the design hierarchy and an abstraction of the design behavior with regard to power/energy usage.

1.2 Purpose

Current low power design and verification standard (IEEE 1801-2013 and IEEE P1801) is focused on the voltage distribution structure in design at Register Transfer Level (RTL) description and below. It has minimal abstraction for time (only interval function for modeling clock frequency), but depends on other hardware oriented standards to abstract events, scenarios, clock trees, etc. which are required for energy proportional design, verification, modeling and management of electronic systems. The necessary abstractions of hardware, as well as layers and interfaces in software are not yet defined by any existing standards. This standard addresses energy proportionality through tight interplay between energy-oriented hardware and energy-aware software. It provides new design, verification, modeling, management and

testing abstractions and formats for hardware, software and systems to model energy proportionality, and enables the design methodology that naturally follows the top-down approach - from the system and software down to the hardware.

1.3 Typographic conventions

The following typographic conventions for the description of the Unified Hardware Abstraction (UHA) syntax definitions are used in this standard.

- The `courier` font indicates UHA code, e.g., `cachesize = 32768;`
- The **bold** font is used to indicate key terms or symbols, text that shall be typed exactly as it appears, e.g., the default object type is **component**.
- The *italic* font represents user-defined UHA variables. For example, in the following line, *propname* and *value* are variables (NB: **=** and **;** are required symbols here).

```
propname [ = value];
```

— Square brackets ([ ]) indicate optional parameters.
— Angle brackets (<>) indicate multiple parameters that need to be used together in the construct.

These conventions are for ease of reading only. Any editorial inconsistencies in the use of this typography are unintentional and have no normative meaning in this standard.

2 Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.[1]

2.1 Definitions

abstract object: An **object** that does not represent an object instance in the final model, but is used as the basis for derived models via inheritance, where the abstract object's properties are inherited by the derived object.

component: An **object** representing a hardware **component**.

control component: A component representing an entity that changes the power state of a hardware component. Control components can be of type `clock`, `reset`, or `voltage-supply`.

event: An **object** describing the occurrence of environmental change or user activity.

fundamental state: A top-level state of a hardware **component** or software **task**.

impact: An **object** describing the hardware impacts triggered by an **event**.

object: A description unit describing a hardware **component**, software **task**, **event**, **impact**, **state**, or transition.

state: An **object** representing a power state of a hardware **component** or software **task**.

root object: An implicit object which is the parent of all top-level **object**s.

task: An **object** describing the properties of software with respect to its resource requirements and power behavior.

2.2 Acronyms and abbreviations

UHA Unified Hardware Abstraction

UPF Unified Power Format

[1] *IEEE Standards Dictionary Online subscription* is available at: http://ieeexplore.ieee.org/xpls/dictionary.jsp.

3 Description principles

This clause describes the underlying principles of UHA.

3.1 Programmer's view

UHA promotes describing a systems power related behavior from a programmer's view. Any component or software task which produces a power effect can be described as an object in UHA.

The UHA description of a hardware *component* contains information to describe its power behavior, as well as the dynamic power behavior on state changes. The software is described as UHA *tasks*, which depend on those hardware components, triggering power state changes. UHA allows users to model the state changes of a system and thereby determine the dynamic power consumption of a system in response to these state changes.

3.2 Modularity and reuse

Descriptions of individual components or of subsystems can be included in other system descriptions without requiring the description of the included component or subsystem to be modified.

3.3 Description elements

The primary element of the UHA description format is the UHA object. *Objects* are used to describe hardware components, software tasks, events, and system impacts. See also Figure 1,

.

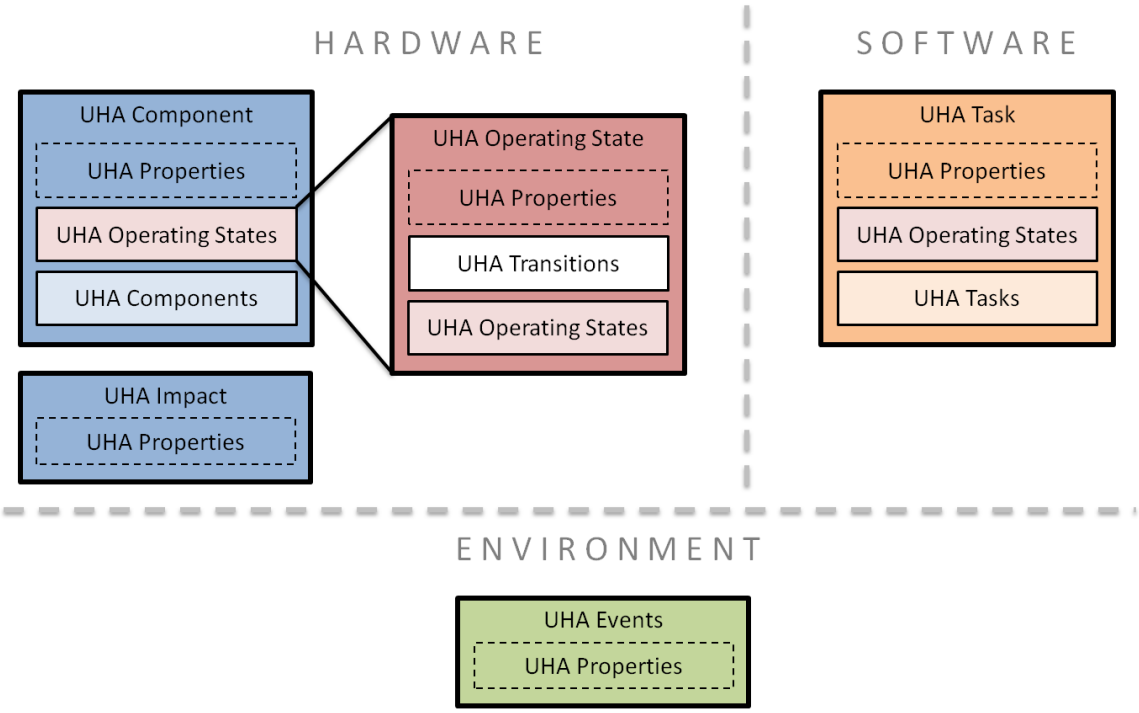


Figure 1—UHA objects

UHA code may contain comments. Single-line comments start with `//` and continue until the end of the line. Multi-line comments start with `/*`and end with `*/`.

4 Modeling basics

This clause describes a few basics about UHA. Subsequent clauses further detail each of these constructs and concepts.

4.1 UHA language elements highlights

The UHA language is based on two high level language elements: objects and properties. The basic syntax of each is described in 4.1.1 and 4.1.3, respectively.

4.1.1 UHA objects

UHA objects are defined using a name followed by braces (**{}**) containing the object body:

```
object-name { …body… }
```

The object body may contain properties followed by other objects, also referred to as sub-objects.

There are different types of UHA objects, the standard type being a **component** (see 6.1). For all other types (see 5.1), a *type qualifier* is used before the name, e.g.,

```
state state1 { …body… }
```

where `state1` defines a state.

4.1.2 Control components

UHA can also be used to explicitly model clocks, voltage rails, power islands, etc. by defining control components (see 8.2).

The following code is an example of doing so for each of the three types of control components.

```
clock clk1 { …body… }

voltage-supply vdd1 { …body… }

reset vdd1 { …body… }
```

4.1.3 UHA properties

UHA properties are defined using a simple assignment: a *name* and a *value* separated by an equal sign (=). Note that a semicolon (;) is mandatory following the property *value*. There are different data types for properties, see 0.

```
name = <property-value>;
```

4.1.4 UHA parameters

Parameters are a type of property used to model properties that can change at runtime (see 5.3.2). Parameters are defined as follows:

```
name = parameter {
        [min = value;]
        [max = value;]
        [values = list_of_values;]
}
```

Either the **min**/**max** fields can be used to define a range or the **values** field can be used to define a list of permitted values.

4.2 Minimal UHA description

Every UHA description shall contain at least one root object, typically using the name of the system:

```
mysystem { …body… }
```

where `body` contains the complete description of the *system*, its components and states, as well as its software and dependencies.

4.3 Managing larger descriptions

Descriptions can be distributed across multiple files, where the `include` directive can be used to concatenate multiple files (see Clause 12).

If a description spans multiple files, each file shall contain a root object, while the same name can be used for the root objects in every file. In fact, it is recommended to use the same root object name across all files of a single system to facilitate hierarchical descriptions and reuse.

5 Elements

A *system* is described in UHA using a set of hierarchically nested objects. One object can describe any entity including entire systems, blocks, components, software tasks, and events. Objects can be nested to establish a hierarchy.

The object body consists of *properties* as well as *child objects*. The following syntax applies for all UHA objects. Any exceptions are to this syntax are noted in the description for that type (e.g., no child objects for **transition**s, see 7.6) and all special properties for each *type* are shown in its corresponding description (e.g., see 6.2 for the **component** ones).

```
[type] objectname {
      [properties]
      [child objects]
}
```

5.1 Object types

UHA distinguishes the following object types:

- **component** [default]
- **clock**
- **reset**
- **voltage-supply**
- **state**
- **transition**
- **event**
- **task**
- **impact**

The default object *type* is **component** (see 6.1), while all other *type*s require their respective type qualifier before the object name, e.g., for a `vdd1` object of type `voltage-supply`:

```
voltage-supply vdd1 { … }
```

5.2 Object names

Object names shall be locally unique, i.e., with respect to all other objects with the same parent. This also applies to top-level objects, i.e., all objects defined without an implicit or explicit parent. Object names, however, are not required to be globally unique.

Object names are strings 1 to 31 characters long. Object names shall include at least one alphanumeric character and may also include dashes (-) and underscores (_). Object names are case-sensitive.

5.3 Properties

The characteristics and relationships of UHA objects are described using UHA *properties*. Such properties are fixed at design time. That is, the value of the UHA property is constant throughout the object's lifetime.

Properties consist of a name and a value.

```
propname [ = value];
```

Examples:

```
cachesize = 32768;

description = "sample object";
```

Property names have the same restrictions as object names, see 5.2. Some property names are reserved for special purposes. The list of special properties is shown in the table below.

Table 1 – *Special Properties*

Property / parameter name	Scope	Defined
capabilities	state	7.3.1
clock-parent	component	6.2.2
clock-state	state	8.6
default-state	component	7.5.4
entry-latency	state	7.5.3
exit-latency	state	7.5.3
frequency	component	8.1
frequency-output	clock	8.3.2
from	transition	7.7.1
latency	state, transition	7.7.2
parent	component	5.4
power	component, state, transition	6.2.1, 7.5.2, 7.7.3
power-output	voltage-supply	8.5.3
requires	state, task	7.4
reset-parent	component	6.2.2
reset-state	state	8.6
voltage	component	8.1
voltage-output	voltage-supply	8.5.2
voltage-parent	component	6.2.2
voltage-state	state	8.6

NB: All *object*s can have unlimited user-defined properties (UDPs), some also have special properties (e.g., see 6.2 for the **component** ones).

5.3.1 Data types for property values

A property value contains the information associated with the property.

The following elementary data types are supported for properties:

— boolean

— numeric (integer or floating point numbers)

— formatted address

- text
- reference

Additionally, UHA also supports complex data types, which combine two or more elementary values:

- value pair
- list
- lookup table
- expression
- composite

5.3.1.1 Boolean

Boolean properties can either be defined using **true** and **false** or they can be defined without an assignment. Boolean properties that evaluate to `true` can also be defined without an assignment.

In the following example, `prop2` and `prop0` have the same value.

```
prop0 = true;

prop1 = false;

prop2;
```

5.3.1.2 Numeric

UHA has two numeric data types: integer and floating point.

Integer values up to `64` bits are assigned as follows:

```
propname = 0;

propname2 = 1000;
```

Values larger than `64` bits need to be broken into multiple words.

Integer properties shall be decimal only. Hexadecimal values are treated as formatted addresses.

Floating point values are assigned as follows:

```
propname = 0.0;

propname2 = 37.5;
```

An integer literal or a floating point literal may contain a suffix as its unit. The suffix shall be one of the values listed in Table 2.

Table 2 —*Units in numeric values*

Unit type	Supported units
Duration	h, min, s, ms, us, ns
Voltage	uV, mV, V
Power	uW, mW, W
Energy	J, mJ, uJ, wH
Current	uA, mA, A
Frequency	Hz, kHz, MHz, GHz

5.3.1.3 Formatted address

Formatted addresses are described using a `0x` prefix and a `.` separator to create groups of four hex digits. Addresses larger than `64` bits need to be broken into multiple words.

```
propname = 0x1;

propname2 = 0xFF00.0000;

propname3 = 0xa0.ffff;
```

5.3.1.4 Text

Text values are enclosed within double quotes (“*text*”).

```
description = "some power object";
```

5.3.1.5 Reference

A reference (**&**) is used when a property refers to another object, property, or parameter in the description. References can be by name only or by a path.

```
propname = &otherobject;

propname2 = &path/to/otherobject;
```

For more details on references in UHA descriptions, see 5.5.3.

5.3.1.6 Value pair

Property values of type `value pair` contain one or more pairs of data, each pair combining two elementary values.

```
propname = <leftval1 rightval1> [, <leftval1 rightval1>];
```

A property containing a value pair is often used to detail how to set a certain configuration register or signal, e.g.,

```
set =  <&cpu/utilization 50>;
```

Value pairs can also include an operator:

```
propname = <leftval1 operator leftvale2>;
```

The following *operator*s are supported:

- **EQ** – equals
- **GT** – greater than
- **GE** – greater or equal
- **LT** – less than
- **LE** – less or equal

Example:

```
requires = <frequency GE 1GHz>;
```

5.3.1.7 List

Lists can be created by separating multiple values using commas (**,**). These values can include any of the elementary property value types described in 5.3.1.

5.3.1.8 Lookup table

A lookup table can be defined as a special property and be referenced in other data types:

```
brightness_level_lut = [dark 20], [medium 48], [light 274];
simple_table = [10, 20, 30, 40, 50];
```

A lookup table can be used in expressions with the operator **[ ]**:

```
brightness_level_lut   [   dark   ]   //   Retrieve   the   value   "20"
                                      //  in the lookup table
```

Here's an example of how a lookup table is referenced from a power model:

```
power = simple_table[3];
```

5.3.1.9 Expression

Properties can be assigned a basic expression. An expression is a formula consisting of operands and operators.

5.3.1.9.1 Operands

Operands can be literals, properties, as well as expressions enclosed in parentheses (**()**).

5.3.1.9.2 Operators

Supported operators are the basic arithmetic operators **+**, **-**, ***, /, as well as parentheses (**()**) and the array/lookup operator (**[]**).

5.3.1.9.3 Examples

The following are some examples of possible UHA expressions.

```
power = basepower + utilization * 0.1;
power = basepower + factor * (utilization + 5);
power = brightness_level_lut[brightness]*0.8+1;
```

5.3.1.10 Composite

A composite property is used to describe complex properties, such as:

- parameters
- capabilities
- energy profiles
- activity profiles

Composite properties have a set of pre-defined fields which can be specified. Composite properties can be nested.

5.3.2 Parameters

A UHA parameter is a composite property that describes a variable attribute of a UHA object, i.e., its value is to be resolved at run time.

```
cpu-utilization = parameter {
      description = "Utilization (percentage) of CPU";
      default = 0;
      max = 100;
      min = 0;
}
```

UHA parameters can only have numeric types: integer or floating point numbers.

UHA parameters may specify the optional fields: **description**, **default**, **min**, **max**, and **values**. **default** defines the default value of the parameter. **min** and **max** limit the minimum and maximum values of the parameter. If the parameter can only be one of multiple discrete values, these values can be defined by the **values** field, e.g.,

```
output = parameter {
        values = 200, 400, 800;
}
```

5.4 Object hierarchies

Object hierarchies are typically established using structural nesting.

```
subsystem0 {
      component0 { … }
}
```

In some cases a parent-child relationship may not be easily expressed using structural nesting, e.g., if the parent and child object are to be described in different UHA files. In those situations, it is possible to explicitly reference the parent through the **parent** special property. The following is an example of an explicit hierarchy using the `parent` property, where the objects `subsystem0` and `component0` are in a parent-child relationship:

```
subsystem0 { … }
object0 {
      parent = subsystem0;
}
```

Semantically, the second description here is equal to the first one.

Each system described by UHA contains one or more top-level objects. The same top-level object can occur in multiple files, in which case, the objects' content is concatenated.

The hierarchies of UHA objects allow each UHA object to have a unique absolute path starting from the root of all top-level objects. A path is specified as a list of names separated by the character /. A path can be an absolute path or a relative path. An absolute path always starts with a /, indicating the starting point of the path is the root. A relative path is assumed to be starting from the current object.

5.5 Abstract objects and inheritance

UHA supports the definition of abstract objects, from which other objects can be derived. Abstract objects help avoid code duplication for multiple objects with common properties or states.

If object `B` derives from object `A`, then `A` is defined as the base object of `B`, while `B` is defined as the derived object of `A`.

An object derived from an abstract object inherits all of its properties, states, and sub-components. The abstract object itself, however, does not become part of the resulting system model. Therefore, an abstract object cannot be directly referenced other than using the **is** keyword when defining an object.

The list of keywords is shown in Table 3.

Table 3 —*UHA keywords*

Keyword	Scope	Defined
abstract	component	5.5.1
capability	property	7.3.1
clock	component	8.3
event	component	Clause 9
frequency	property	8.1
impact	component	10.3
is	component	5.5.2
parameter	property	5.3.2
reset	component	8.3.3
state	component	7.1
task	component	10.1
transition	component	7.6
voltage-supply	component	8.5
voltage	property	8.1

5.5.1 Defining abstract objects

To define an abstract object, the object name is followed by the **abstract** keyword.

The following example defines an abstract object named `arm9processor` with two states, `off` and `on`.

```
arm9processor abstract {
    cpu;
    state off {
          power = 0mW;
                      entry-latency = 16us;
          }
    state on {
          power = TBD;
                      entry-latency = 36us;
```

```
                }
        }
```

For abstract objects, properties and parameters can be defined with a value of **<TBD>**, which means the value and type of the property need to be defined in a derived object. An object that inherits from an abstract object shall override all properties having a `<TBD>` value with a concrete value, unless the inherited object itself is also an abstract object.

5.5.2 Defining derived objects (directly)

In order to define an object derived from an abstract object, the **is** keyword is used followed by a reference to the abstract object:

The following example defines an object named `cpu0`, which inherits the properties and states defined in the abstract object `arm9processor`.

```
cpu0 is &arm9processor {
}
```

Derived objects may also be abstract, enabling multiple levels of inheritance, e.g.,

```
arm9processor abstract is &armprocessor {
    endian = little;
}
```

A derived object can override the definition of any inherited property, parameter, or state by defining a new one with the same name.

5.5.3 Including objects as abstract

UHA files can be included by using `/include/ abstract` (see Clause 12) which turns the included model into an abstract object…

5.6 References to an object

A reference (**&**) is used when a property refers to another object, property, or parameter in the description. References can be by name only or by path.

A path typically contains the character **/** as separators. When it is possible for the notion **/** to create an ambiguity with the division operator **/**, a path can be surrounded by a pair of curly braces (**{}**). A `/` character inside `{}` shall never be treated as a division operator.

Here are two examples.

```
propname = &otherobject;

propname2 = &path/to/otherobject;
```

The first example shows a reference by name, referring to the first occurrence of a component with that name. References by name can result in ambiguous descriptions if the name is not unique within the UHA description.

The second example shows a reference by path, referring to the first object with that path. If the path is complete, i.e., starting from the root level, there is no risk of ambiguity. If the path is a partial path, then there still is the possibility of ambiguity: if a description contains two identical sub-paths within different branches.

A reference property is resolved by using the following sequence. It shall be an error if a reference does not match any object after following this sequence.

a) A property or parameter of the current object.

b) A property or parameter of an object from which the current object inherits. Preference shall be given to the nearest ancestor in the inheritance tree if more than one match is found.

c) Another object which is a direct or indirect parent of the current object in the object hierarchy tree.

d) Preference shall be given to the nearest parent if more than one match is found.

One or more reference properties shall not create circular references. The following two examples are grammar errors.

Error Example 1:

```
propname = &propname;
```

Error Example 2:

```
propname1 = &propname2;
propname2 = &propname3;
propname3 = &propname1;
```

6 Hardware components

It is important to model the device's individual components for accurate modeling of the power consumption of an electronic device. Because the components usually consume different amounts of power when they are running in different states, it is equally important to model their states.

6.1 Component definitions

UHA objects of type **component** are used to describe any kind of hardware component.

Components in an electronic device are naturally ordered in a tree hierarchy, with the device itself as a top-level object, `any` components as its children, and `any` sub-components as the children of the components. Here is an example showing an APU and its two CPU cores:

```
soc {
    apu{
      cpu0 {
      }
      cpu1{
      }
}
```

A **component** typically also defines **state**s, see 7.1.

6.2 Special properties

Hardware components can define the following special properties.

6.2.1 power

The **power** property specifies a power model for a **component** (see 6.1), defining how that object's power consumption is computed. A power model may reference run time parameters through an expression or via an external UPF or C function.

Specifically, power models can be expressed as:

- constant values
- arithmetic expressions
- references to external power models (e.g., specified in UPF or C)

Power models specified on the component level are superseded by power models specified in the current state of the component. If no state-specific power model exists for the current state, then the power model defined here is the one that is applied. See 7.5.2 for more details on state specific power models.

6.2.2 voltage-parent, clock-parent, and reset-parent

The special properties **voltage-parent**, **clock-parent**, and **reset-parent** are used to establish special parent-child relationships between a control component and a hardware component or another control component.

See Clause 8 for details on control components.

6.3 Capabilities

Capabilities are defined on the component level and specify the constraints for the capabilities referenced in the states. A *capability* defines a particular capacity that one or more **state**s of the **component** (see 6.1) may have. Similarly, at a state level, special property **capabilities** can be defined to declare the capabilities that the **state** has, matching one or more **capability** properties declared at the component level. See 7.3.1.

6.4 Power models of hierarchical components

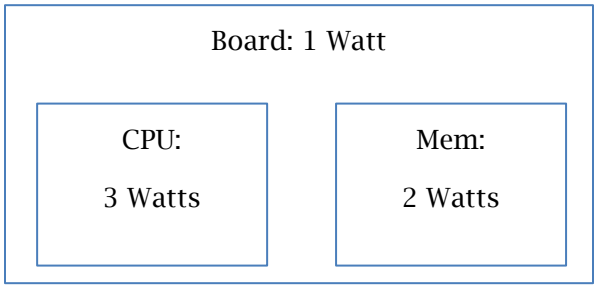
Power models are typically specified for leaf components. However, parent components may also include a power model. The power model of the parent shall only model the incremental power of the parent (accounting for interconnect or not modeled components), unless the subcomponents do not include their own power model.

In order to illustrate the mechanics, consider a simple system hierarchy as follows:

Board: 1 Watt

CPU: 3 Watts

Mem: 2 Watts



The total power consumed by the board and its subcomponents is the sum of the above, i.e., 6 Watts. However the power model of the board will only account for the components which are not explicitly modeled, i.e., it will not include the 3 Watts of the CPU or the 2 Watts of the memory.

The following source code shows the corresponding UHA description.

```
board {
      power = 1W;
      cpu {
            power = 3W;
      }
      mem {
            power = 2W;
      }
}
```

On the other hand, it is also possible to model the power entirely on the level of the parent, e.g., as follows:

```
board {
      power = 6W;
      cpu {
      }
      mem {
```

```
            }
        }
```

In this UHA version, neither `cpu` nor `mem` have their own power model; hence, the board's power model now accounts for the power of the cpu and the memory component as well.

7 States and transitions

UHA states (see 7.1) describe different power states providing different sets of capabilities for a single component or a set of components. Any **component** (see 6.1) can have one or more **state**s. States can be nested to form hierarchies, e.g., `inactive/idle` vs. `inactive/off`.

UHA transitions can be explicitly defined to model the power-profile, latency, and behavior of state transitions. See 7.6.

7.1 States

The possible power states a component can be in are described using UHA states. A **state** is always a child object of a **component** (see 6.1) and does not exist independently of components.

NB: States can only have **state**s or **transition**s as child objects.

A **state** can specify:

a. its own power model
b. its capabilities
c. its requirements including
    1. frequency and voltage
    2. other components' states or capabilities
d. its entry and exit latency

The following example shows the description of a simple peripheral with two states, `active` and `inactive`.

```
peripheral {
      state active {
            power = 20mW;
            capabilities = receive, transmit;
            requires = <frequency 384kHz>;
      }
            state inactive {
                  power = 1mW;
      }
}
```

7.2 State hierarchies

It is possible to model the **state**s of a **component** (see 6.1) in a hierarchical manner. For example, a CPU core may have the states `off`, `idle`, and `awake`, where the `awake` state may have sub-states reflecting the CPU core's multiple Dynamic Voltage and Frequency Scaling (DVFS) states.

If a state is not a sub-state of any other state, it is considered a *fundamental state*. The states of a component form a tree rooted in the parent of all fundamental states. As the example shown in Figure 2 indicates, a CPU core may have an `awake` state which may further have its sub-states indicating their frequencies.

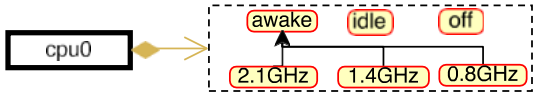


Figure 2—Tree of states of a CPU core

If a component is in a sub-state, it is also automatically in the corresponding parent state(s). In the example in Figure 2, if the CPU core is in the state of `2.1GHz`, it is inferred that it is also in the `awake` state.

If both the parent state and the sub-state(s) specify power models, the power model closest to the leaf of the state hierarchy supersedes the power model(s) from the parent states.

For every component, only one power model is selected at any given time. If a power model is specified on the component level, this model is chosen only when no state specific model applies.

7.3 Capabilities of components and their states

Capabilities are properties of **state**s, e.g. a memory can be capable of reading and writing while active. Capabilities are used as a proxy to express requirements related to a component's state, to avoid having to explicitly require specific states.

Capabilities can be defined implicitly when referenced in a **state**. In the following example, the capabilities `read` and `write` are defined implicitly as shared Boolean capabilities.

```
ssd-memory {
            state active {
            capabilities = read, write;
            }
            …
}
```

7.3.1 Capability definitions

A capability can be defined explicitly in a component using a composite property (see 5.3.1.10):

```
capabilityName = capability {
      type=[shared | additive | control];
      min=;
      max=;
      values=;
}
```

Capability names (e.g., `capabilityName`) are completely user-defined. While there are implicit semantics, the actual name could be `bandwidth` even if it defines a *throughput* capability.

The value of a capability property may contain the optional fields: **type**, **min**, **max**, and **values**. If a capability does not define any of these optional fields, the **capability** definition at the **component** level (see 6.1) can be omitted.

The states may also specify a property's **capabilities** listing the capabilities this component has when it is in this **state**:

```
state name {
        capabilities = […];
    }
```

7.3.2 type

The **type** field of a **capability** definition (see 7.3.1) can have the values **shared**, **additive**, or **control**. If the **type** property is omitted, it will default to **shared**.

A capability with a **shared** type indicates this capability can be shared among different requirements. For example, for the component `ssd-memory` declared in the code listing in 7.3, if two different tasks require the component to have the capability `read` at the same time, this is not an error since the `read` capability has the default type `shared`.

A capability with an **additive** type indicates this capability is a numeric value that needs to be added as different requirements are made to the component. Examples are a `bandwidth` or `qos` capability:

```
eth0 {
    readPercentage = capability {
      min = 0;
      max = 100;
    }
    bandwidth = capability {
        min = 0;
        max = 1000;
        type = additive;
    }
    link = capability {
    }
    state max {
        capabilities = link, <bandwidth 1000>;
    }
    state mostlyread {
      capabilities = <readPercentage 80>;
      power = eth0power(readPercentage);
    }
    state lowpower {
        capabilities = link, <bandwidth 100>;
    }
    state off{ // no capabilities declared
    }
}
```

A capability with a **control** type is reserved for voltage-output and frequency-output capabilities of clock and voltage-supply components. See Clause 8 for more details on control components.

7.3.3 min and max

For numeric capabilities, such as `bandwidth`, the special properties **min** and **max** can be used to define the minimum and maximum possible values of the capability respectively.

7.4 State dependencies

Dependencies can be expressed either as a dependency on other components' capabilities or by referencing specific **state**s of other **component**s (see 6.1).

The property **requires** is used to specify this component requires another component to provide a certain capability (see 7.3) or to request a specific state for another component:

```
requires = <&referenced_component/capability OP value>;

requires = &referenced_component/state;
```

If a **state** has multiple requirements, they can be specified by either using multiple **requires** properties or **set** properties (see 7.5.1) or by using a *list*, i.e. separating multiple requirements with commas (**,**).

7.4.1 Asserting capability requirements

Capability requirements can reference both shared capabilities and additive capabilities.

7.4.2 Requiring a shared capability

In the Ethernet example shown in 7.3.2, the requirement:

```
requires = &eth0/link;
```

maps to any state that provides the `link` capability, i.e., `max` or `lowpower`.

7.4.3 Requiring an additive capability

The **requires** property can express a dependency on an additive numeric capabilities by specifying a comparative operator:

- **EQ** – equals
- **GT** – greater than
- **GE** – greater or equal
- **LT** – less than
- **LE** – less or equal

The following requirement states that the `bandwidth` needs to be at least `80` to be fulfilled:

```
requires = <&eth0/bandwidth GE 80>;
```

Based on the Ethernet component example shown in 7.3.2, the `bandwidth` requirement expressed therein can be met by either the `max` or `lowpower` state.

If another requirement is added on top of this requirement (e.g., due to a different task running in parallel):

```
requires = <&eth0/bandwidth GE 30>;
```

then the new requirement and the existing requirement are added together, which means the `bandwidth` requirement has increased to `110Mbps`.

When one or more requirements are placed on a **component** (see 6.1), it shall be a run time error if no **state** can meet the requirements. When multiple states can meet all of the requirements requested of the component, it is presumed the component will transfer to the state with the lowest power while still meeting all the requirements.

7.5 Special properties of states

The following properties are used to define what happens when a **state** is entered.

— **requires** (dependency) – requires another **component** to provide a capability. See 7.4.

— **set** – sets the value of a parameter.

— **power** – power model for the component in the current state.

— **entry-latency and exit-latency** – latency values to enter or leave the current state.

— **default** – indicates the state is the initial (default) state.

All of these special properties are optional.

7.5.1 set

The **set** property sets the value of a parameter.

7.5.2 power

The **power** property of a **state** defines a power model specific to that **state**. The syntax for defining the **power** property of a **state** is the same as for the **power** property for a **component** (see 6.2.1).

At any given time, a **component** has exactly one active power model. The selection of this power model is based on the current **state** of the **component**.

When the **state** is active and both the **state** and the **component** itself have their **power** property defined, the **state** specific power model dominates.

Similarly, if there is a hierarchy of states defined, the power model of the leaf always trumps the power model of the parent(s). During a transition, a power model specified in the transition object trumps all other power models for that **component**.

7.5.3 entry-latency and exit-latency

A state's special properties **entry-latency** and **exit-latency** are numeric latency values to enter or leave the **state**.

7.5.4 default

The **state** the **component** is in initially is identified by the Boolean **default** property. Only one of the **state**s can have the **default** property defined.

7.6 Transitions

In many cases, declaring **state**s may provide enough information to model the power behavior when the components transition between states. However, there are more complex cases where the run time behavior of state transitions depends on both the old and new states. In this case, **transition** objects may be defined to model these transitions.

NB: Transitions cannot have child objects (however, like any other object they can have unlimited user-defined properties).

A *transition object*, if defined, is a child of a state object. It may contain a **from** property indicating the source state (or states) of the transition.

The following is an example of a **transition**:

```
crypto {
    state active {
        transition {
            from = &inactive;
            latency = 80us;
```

```
        power = 2mW;
          }
          transition {
              from = &off;
              latency = 200us;
        power = 1mW;
          }
      }
      state inactive {
          transition {
              from = &active;
              latency = 20us;
          }
      }
      state off {
      }
  }
```

A **state** may contain multiple transitions as its children, each defining different transition paths to this **state** as a target. If a **transition** property does not have a **from** property, the transition object describes the default behavior for transitions. This default behavior corresponds to a transition from any state other than the ones already described in other transition objects under the same target state.

If no transition object is defined for a **state**, it is implied that transitions into this state are possible from any source state. Once at least one **transition** object is defined, then all legal source states need to be covered by transitions, i.e. if a particular state is not listed in the **from** property of a transition and if no generic transition object (a **transition** object without a **from** property) is defined, then a **transition** from that **state** shall not be allowed.

7.7 Special properties of transitions

For the duration of the transition, as specified by the **latency** property (see 7.7.2), a power model specified in the transition object trumps all other power models for that **component** (see 6.2.1).

7.7.1 from

If a **transition** depends on the prior state of the component, the **from** property is used to specify the reference to the prior state or states. When a transition applies to multiple source states, the states are enumerated using a comma separated list.

```
from = &state1 [, &state2, &state3, ...];
```

The **from** property is optional. If not specified, the transition applies regardless of the source state.

7.7.2 latency

The **latency** property specifies the duration of the **transition** (the length of time required to transition from the previous state to the new state). This transition-specific latency overrides any value specified in the **entry-latency** or **exit-latency** properties defined on the state level (see 7.5.3).

7.7.3 power

The **power** property is used to describe the power model during the transition time. It is defined as the average power used during the time of the **latency** property (see 7.7.2).

8 Control components

A *control component* is a special type of hardware component used to model the impact of clock, voltage, and reset lines on the power states of an ordinary hardware component. Each hardware **component** (see 6.1) may specify a parent of each type. A component's **state**s (see 7.1) may depend on the **state** or value of its control parents.

8.1 Modeling voltage and frequency

Voltage and frequency can be modeled without defining control components explicitly. Each component is assumed to have a voltage and a frequency, coming from an ideal power source and an ideal clock. The required clock frequency for a state can be expressed by specifying requirements on the **frequency** property and the required voltage can be expressed defining a requirement on the **voltage** property.

Ideal clocks can assume any frequency and ideal clocks are independent of each other. Ideal voltage domains can supply any voltage to any component without any dependencies.

The properties **frequency** and **voltage** are essentially reserved reference properties that are implicitly referencing the respective control capability of their control parents:

```
frequency = &clock-parent/frequency-output;
voltage = &voltage-parent/voltage-output;
```

If such control parents are not assigned, it is presumed the component has an implicit "ideal" clock-parent or voltage-parent.

Control components can also be used to specify constraints of the individual clock components and voltage domains and power islands in expressing the hierarchical nature of clock trees and voltage domains.

8.2 Declaring control objects

By specifying a component-type modifier before the component name, a component can be declared to be one of the following control components:

- **clock** – representing a clock or clock source/PLL
- **voltage-supply** – representing a voltage rail, power domain, or power island
- **reset** – representing a reset line

The following is an example of a **clock** component:

```
clock reference_clk {
    frequency-output = capability {
        type = control;
        min = 200MHz;
        max = 1200MHz;
    }
}
```

8.3 Clocks

Clock objects are control components representing clocks or clock sources, providing a clock signal to connected components. Clocks have a frequency associated to them, modeled as a parameter.

8.3.1 States

By default a **clock** has two states: **ON** and **OFF**. If the user does not explicitly define these two states, they are implicitly defined by the following logic: In the `OFF` state the frequency is zero (`0` [clock gated]), while in the `ON` state the clock has the frequency that was last set.

Example:

```
clock a-clock-node {
      clock-parent = &another-clock-node;
      state ON { … }
      state OFF { … }
}
a-peripheral {
      clock-parent = &a-clock-node;
      state active { clock-state = ON; }
      state idle { }
}
b-peripheral {
      clock-parent = &a-clock-node;
      state active { clock-state = ON; }
      state idle { }
}
```

8.3.2 Frequency-output capability

All **clock**s have a capability called **frequency-output**. Unless the frequency is driven by its own clock parent, a clock shall explicitly define **frequency-output** to constrain the supported frequencies for that clock. Frequency values can either be specified in `Hz` or using an explicit unit suffix, such as `kHz`, `MHz`, or `GHz`.

The following example constrains the clock frequencies to range from `100MHz` to `1500MHz`.

```
frequency-output = capability {
      type = control;
      min = 100MHz;
      max = 1500MHz;
}
```

Alternatively, it is also possible to specify a discrete set of permitted frequencies using the **values** property.

The following example specifies the supported frequencies explicitly. Similarly a fixed clock can be described using a single frequency in the `values` field.

```
frequency-output = capability {
      type = control;
      values = 200MHz, 333MHz, 500MHz, 1000MHz;
}
```

8.3.3 Modeling clock dividers

Clock dividers can be modeled by specifying the values field of the frequency-output capability using an expression. This expression specifies the relationship to the input frequency, i.e. the frequency supplied by the clock-parent.

```
frequency-output = capability {
```

```
        values = expression(frequency);
}
```

Common examples of such values expressions are:

```
values = frequency;       // no divider (just a clock gate)
values = frequency / 2;   // fixed divider
values = frequency / div;// div is a parameter
```

When a clock component does not explicitly define a frequency-output capability, the frequency-output is assumed to equal the frequency of the clock parent.

8.4 Resets

Reset objects are control components representing reset lines used to hold components or subsystems in reset.

Reset objects have the states **ASSERTED** and **RELEASED** by default.

8.5 Voltage-supply

Voltage-supply objects are control components representing voltage rails and/or power domains, also referred to as *power islands*. Voltage-supply objects have an output voltage level associated with them, modeled as an explicitly defined capability.

8.5.1 States

Voltage-supply objects have the states **ON** and **OFF** by default. In the `OFF` state, the voltage level is zero (`0`), while in the `ON` state, the voltage level is the value last set.

8.5.2 Voltage-output capability

The voltage-output capability need to be defined to delineate the supported voltage range, unless the voltage is provided by a voltage-parent, e.g.,

```
voltage-output = capability {
        type = control;
        min = 800mV;
        max = 1300mV;
}
```

8.5.3 Power-output capability

The power-output capability is an implicit capability representing the power supplied by a voltage supply to its connected components. The power-output capability may be explicitly specified to describe constraints, such as the maximum power supported by the voltage supply.

The power-output capability may also be referenced in the power model of a voltage supply to express the power consumption as a fraction of the power consumed by the connected components, e.g.,

```
power = 0.2 * power-output;
power = (1 - efficiency(power-output)) * power-output;
```

Here, the first power model uses a simplistic static efficiency to compute the power consumption of the voltage supply, while the second power model uses a C-function to determine the efficiency of the power supply based on the power consumption of the connected components.

8.6 States driven by control components

If a **component** (see 6.2.1) has defined its relationship to one or more control components through the special properties **voltage-parent**, **clock-parent**, or **reset-parent**, it is possible to define state requirements related to those control components. Requirements on the component's control components can be expressed by referencing the state required of the control component or by using the special properties **voltage-state**, **clock-state**, and **reset-state** to express a bi-directional dependency.

8.6.1 Basic control component dependencies

When referencing a control component's state directly from a requires property, the requirement is considered unidirectional, i.e. a basic dependency.

In the following example, `vdd1` needs to be in its `ON` state for component `C` to enter state `S0`. However, the reverse dependency is not true, i.e., `vdd1` may enter its `ON` state without component `C` entering state `S0`.

```
C {
    voltage-parent = &vdd1;
    state S0 {
      requires = voltage-parent/ON;
    }
}
```

8.6.2 Bidirectional control component dependencies

A bidirectional dependency between a component and a control component is expressed using the special properties **voltage-state**, **clock-state**, and **reset-state**.

In the following example, `vdd1` also needs to be in the `ON` state in order for `C` to enter `S0`. Additionally this description also implies whenever `vdd1` enters its `ON` state, component `C` entesr its `S0` state.

```
C {
    voltage-parent = &vdd1;
    state S0 {
      voltage-state = ON;
    }
}
```

The following example defines three states, each mapped to a different combination of states of its control components.

```
cpu0 {
    voltage-parent = &apu_pwr;
    clock-parent = &apu_clock;
    reset-parent = &apu_reset;

    state ON{
        clock-state = ON;
        voltage-state = ON;
        reset-state = RELEASED;
    }
```

```
        state IDLE{
            clock-state = OFF;
            voltage-state = ON;
            reset-state = ASSERTED;
        }
        state OFF{
            clock-state = OFF;
            voltage-state = OFF;
            reset-state = ASSERTED;
        }
    }
```

8.7 Dependencies on frequency and voltage

Dependencies on **frequency** and **voltage** (see 8.1) can be expressed using the **requires** property (see 7.4).

The following example defines a state that requires a voltage of `1.1V`, i.e., it requires the component's voltage parent to be in a state which is capable of providing a voltage of `1.1V`.

```
        state maxspeed {
            requires = <voltage EQ 1.1V>;
        }
```

The following example defines a state that requires a clock frequency of `1 GHz` or higher. If a clock-parent is specified, this requirement asserts that the component's clock parent needs to be in a state which is capable of providing such a frequency.

```
        state execute {
            requires = <frequency GE 1GHz>;
        }
```

9 Events

Events are objects describing the occurrence of environmental change or user activity, such as a button press, an application launch, or a signal drop. Events are typically detected via interrupts or register polling.

Instances of event objects can be generated by designers during initial what-if-analysis, by simulators at different levels of abstraction, by emulators, by the probes of test and measurement equipment, or by the event observation and detection logic executing in real-time on the running device.

Properties of events are the detection logic, the event source, and the repeat cycle.

```
event interrupt_event {
      detection_logic = ; //  function detecting the event
      source = ; // description of the event source
      repeat = ; //predicted repeat cycle for the event
}
```

Events can describe both future events (e.g,. as simulation input specification) or past events (e.g., detected by the observer logic of the running device) in relation to the time instant in which the event occurred.

10 Tasks and impacts

Tasks and impacts are abstract objects describing the influence of the software and environment on the energy behavior. See Figure 3.

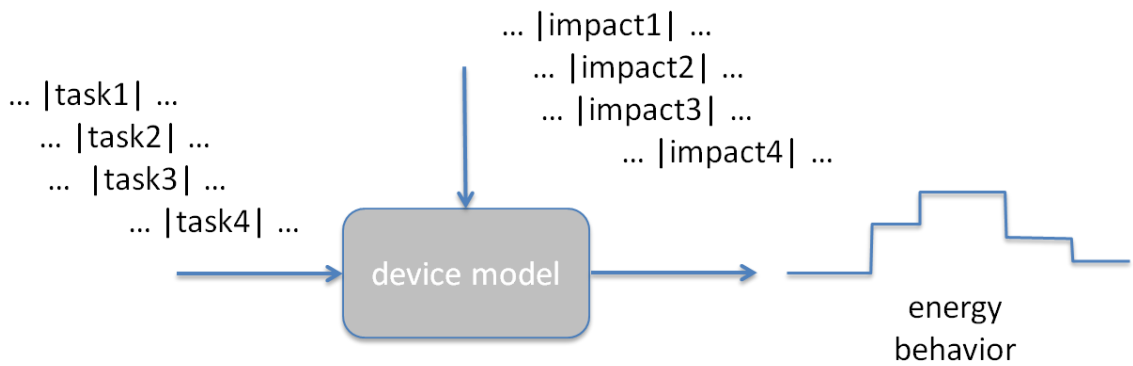


Figure 3—Tasks and impacts

10.1 Tasks

Tasks represent the influence of the software execution on the energy behavior. The software modelled with a **task** could represent a set of instructions (as in interrupt service routines), a system function of an operating system, a complete application, or a complete operating system.

General properties of **task**s are the **begin** event, **end** event, and the expected **duration**, e.g.,

```
task ftp {
      begin = &ftp_started;
      end = &ftp_ended;
      duration = …; //
}
```

The **power** property of **task**s describe the incremental power consumed by the **task**, e.g.,

```
task ftp {
      power = 30mW;
      entry_latency = 28ms;
}
```

Requirement attributes of **task**s are the components needed for execution; specifically, which states or capabilities are required, e.g.,

```
task ftp {
      begin = &event1;
            end = &event2;
            requires = <&eth/performance maxspeed>;
            requires = &usb/active;
}
```

Tasks can have **state**s (see 7.1), which can describe the state of a background task such as a service or an operating system, e.g.,

```
task linux {
```

```
            begin = &linux_boot;
            end = &linux_shutdown;

            state active {
                requires = &cpu/active, &ddr/active;
            }
            state sleep {
                requires = &cpu/sleep, &ddr/active;
                entry-latency = 2ms;
                exit-latency = 15ms;
            }
            state suspended {
                requires = &cpu/suspended;
                requires = &ddr/retention;
                entry-latency = 30ms;
                exit-latency = 120ms;
            }
    }
```

Tasks can be hierarchical, i.e., sub-tasks can be specified.

10.2 Special properties for tasks

The properties described in this section are specific to **task**s.

10.2.1 begin and end

Events (see Clause 9) which trigger the start and end of a task can be represented by the **begin** and **end** properties.

10.2.2 duration

If the task duration is fixed, the **duration** property can be used to specify the length of time the task is expected to run. This property may not be specified in conjunction with the **end** property (see 10.2.1).

10.2.3 requires

Task requirements (**requires**) are typically expressed by referencing component capabilities or transition latency requirements. This allows the resulting power management control software to be more aware of the work that is to be performed and to adjust the component power states accordingly. By using capability requirements, the energy management algorithm has the flexibility to adjust the states of all the components so they match the requirements of the currently running tasks while minimizing the overall energy consumed by the system.

Tasks requirements may also be expressed by referencing states of components explicitly.

10.2.4 set

A **task** shall use the **set** property to specify parameter changes triggered.

10.2.5 power

The **power** property specifies the incremental system power consumed by the **task**, accounting only for the power increase not captured by the power consumption of the affected components and their respective state changes due to the task's requirements. For more details on specifying power models, see 6.2.1.

10.3 Impacts

Impacts are used to capture the influence of the environment or user activity on the system. An **impact** contains properties defining the start/end event, the duration, and the impact behavior.

```
impact name {
        begin = event_reference;
        end = event_reference;
        requires = state_transition;
        requires = …
        set = parameter_change;
        set = …
}
```

The following is an example of a UHA impact:

```
        impact lostsignal {
        begin = &signal_lost;
        requires = &gsm/active/searching_signal;
    }
```

10.4 Special properties for impacts

The properties described in this section are specific to **impact**s.

10.4.1 begin and end

An **impact** shall reference one or more events that trigger the impact using the **begin** property.

An **impact** may also specify an **end** property, identifying the event that causes the impact to cease. If no **end** event is specified, the **impact** is presumed to have a fixed duration.

10.4.2 duration

An **impact** may specify a **duration**. If no **duration** and no **end** event (see 10.4.1) are specified, the **duration** defaults to zero (`0`), i.e. the impact is assumed to be momentary.

10.4.3 requires

An **impact** shall use the **requires** property to specify the requirements triggered when the impact occurs. Typically, this is a state transition of at least one component.

10.4.4 set

An **impact** shall use the **set** property to specify parameter changes triggered.

10.4.5 power

The **power** property specifies the incremental system power consumed by the **impact**, accounting only for the power increase not captured by the power consumption of the affected components and their respective state changes induced by the **impact**.

For more details on specifying power models, see 6.2.1.

11 Power accounting

Component power information is defined on multiple levels. To compute the total power consumption for components and systems, a simple three-step process needs to be followed.

a. The first step is to select a power model for each component.

b. The second step is to compute the power for each component.

c. The third step is to aggregate the total power of the system or sub-system of interest.

The subsequent sections explain each of the steps in more detail.

11.1 Power model selection

Each component shall have one and only one power model at any instance in time. The rules for selecting the active power model based on its current state are described in 6.2.1 and 7.5.1.

11.2 Power value calculation

For each component, its power value is determined using the selected power model and the model's parameter values.

11.3 Power value aggregation

The total power is aggregated hierarchically. Hence, for a given component or (sub)system, its total power is computed by adding the power of all children to its own power value. Finally, the power values of the active tasks and impacts related to this (sub)system are also added.

For example, to account for the total power consumption of the system, the power values for all the components and their sub-components, the power values of the active segment of the energy profile of the running tasks, and the power values of the active impacts need to be added.

11.4 Examples

Figure 4 and Figure 5 show an example of power accounting and consumption for a system in different states. The power model for `component` is selected based on its state, or the transition if the state is in the transition and the transition has power property defined. Then the total power consumption needs to be added to its sub-component.

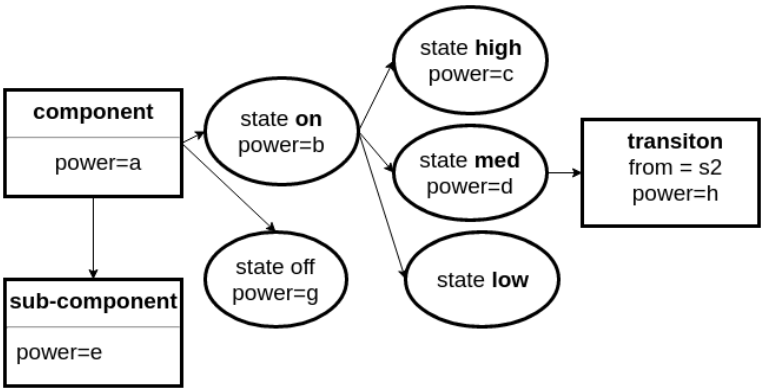


Figure 4—Power accounting

Component state	*Total power consumption*
high	c+e
med	d+e
low	b+e
off	g+e
Transition from “off” to “med”	d+e
Transition from “off” to “high”	c+e

Figure 5—Power consumption

12 UHA file structure

The UHA description of a system is kept as UHA source codes. UHA source codes are organized as a collection of text files with the extension `.uha`, which are referred to as UHA files. A UHA project contains one and only one UHA file as its top-level UHA file.

A UHA file may include other UHA files by using an `include` directive, e.g.,

```
/include/ "filename.uha"
```

A UHA file included by the top-level UHA file can also include other UHA files. A UHA file is part of a project if and only if it is the top-level UHA file, or is included (directly or indirectly) by the top-level UHA file.

The included UHA file can be referred to by its absolute path or a relative path. If it is referred as a relative path, to locate the file on the filesystem, a name searching shall give preference to the UHA file in the same directory as the UHA file being parsed before searching any system-defined directories.

A UHA file may contain an optional annotation of `/readonly/` as a separate line, which indicates this UHA file shall not be edited by the user.

```
/readonly/
```

The layout of a sample UHA project with six files is shown in Figure 6. The top-level UHA file is `file1.uha`.

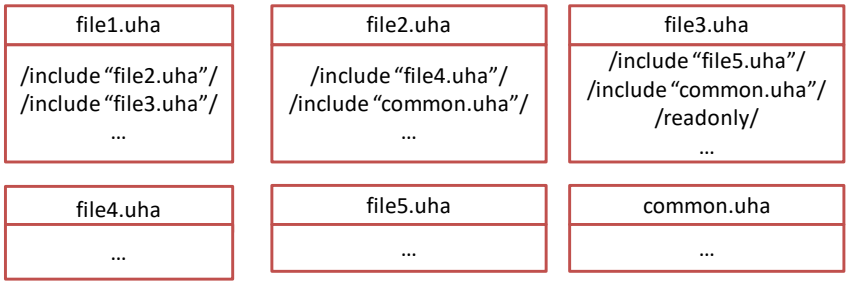


Figure 6— Sample UHA project with six files

An `include` directive may be followed by the **abstract** keyword. In this case, all objects in the included file are treated as abstract objects. These objects are only exposed to the current file, and thus can only be inherited in the current file. For example:

```
/include/abstract "filename.hua"/
```

Annex A

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] IEEE Std 1801™-2015, IEEE Standard for Design and Verification of Low-Power Integrated Circuits.

Revision History

Rev	Change	Date	Author
	9/8/2016 11:59:35 AM Yefu Wang Manual Merge		
	09/08/2016 Yefu Wang promote parameters to a subsection		
	9/8/2016 Davorin Mista, reworked Events, Tasks, Impacts		
	9/8/2016 Yefu Wang Merged with Davorin's changes		
	9/8/2016 5:12pm Change requirements to requires, and other minor changes		
	9/8/2016 10:45pm Remove Scenes, review and cleanup, node=>object, introduce composite property		
	9/9/2016 Include/review Josh's comments		
	9/11/2016 DM: Rewrite introduction to operating states		
	9/14/2016 10:59:18 AM Yefu Wang: activity profile and energy profile, capability mapping, parameter mapping, remove "include as"		
	9/14/2016 4:58:18 PM Yefu Wang: Lookup table		
	9/15/2016 11:03:10 AM Yefu Wang, Power accounting		
	9/15/2016 4:49:03 PM Yefu Wang, parameter / capability map to multiple targets		
	9/15/2016 4:50:57 PM Yefu Wang Minor change in parameter of control components.		
	9/16/2016 1:37:58 PM Yefu Wang Merge changes in power accounting		
	9/16/2016 3pm DM: Explanation of power model selection for transitions, updated UHA object types diagram, updated power accounting/power model selection descriptions		
	9/24/2016 DM: operating-state renamed to state, control node section moved, control nodes are now different object types		
	9/28/16 DM: voltage/frequency are now "capabilities" of control components.		

	10/5/16 DM: use “requires” for ALL requirements including states		
	10/20/16 DM: final review		
	10/21/16 Yefu: P20: Some edits. P28: operating-state -> state		
	10/22/16 Incoporate Josh’s feedback, added proposed abstract/keywords		
	4/25/17 Added power -output capability		
	5/31/17 Adjusted units, clarified frequency/voltage properties		
0.1	moved from IEEE template format to standard Aggios template	6/8/17	dmista
0.2	Joe’s D2 editing pass	6/22/17	jmd
0.3	Joe’s D3 editing pass	6/29/17	jmd
0.4	Joe’s D4 editing pass	7/13/17	jmd
0.5	Added details about bidirectional versus basic dependencies for control components (section 8.6)	7/24/17	dmista
0.9	Removed energy-profile and activity-profile	9/5/17	dmista

References

UHA Modeling Guide – Revision 1.2, 6/20/2017

APPENDIX B: UHA Modeling Guide

UHA Modeling Guide

System Power Modeling with UHA

Version 1.1

11/20/2018

Davorin Mista

Table of Contents

1 About This Document 7

2 Modeling Basics 8

2.1 UHA Modeling Introduction 8

2.1.1 Modeling hardware components and power states 8

2.1.2 Modeling software and external events 9

2.1.3 Modeling dependencies 9

2.2 UHA Syntax 101 10

2.2.1 UHA Objects 10

2.2.2 UHA Control Components 10

2.2.3 UHA Properties 10

2.2.4 UHA Parameters 11

2.3 Minimal UHA Description 11

2.4 Managing larger descriptions 11

3 A Simple System – Step-by-Step 13

3.1 Adding our first component 13

3.2 Adding states 13

3.3 Adding a dependency 13

3.4 Adding power expressions 14

3.5 Adding frequency and voltage 15

3.6 Adding an explicit clock 16

3.7 Adding an explicit voltage-supply 16

3.8 Modeling software tasks and events 17

4 Modeling Dependencies using Capabilities and Latencies 18

4.1 Example: Memory with 3 states 18

4.2 Defining capabilities 18

4.3 Specifying capability requirements 19

4.4 Defining numeric capabilities 19

4.5 Dependencies on numeric capabilities 20

4.6 Latency requirements 20

5 Modeling Voltage and Clock Trees 22

5.1 Modeling frequency and voltage per component 22
5.2 Modeling clocks and voltage components explicitly 22
5.2.1 Components with multiple clocks or voltage rails 23
5.2.2 Modeling clock gates and power gates 23
5.2.3 Modeling clocks without clock-gating or power-gating capability 24
5.2.4 Modeling clock dividers 25
6 Modeling Latencies 27
6.1 Defining entry and exit latency properties 27
6.2 Defining latencies within a transition 28
6.3 Mixing explicit transition latencies and entry/exit latencies 28
7 Modeling Software 30
7.1 Differences between software and hardware components 30
7.2 Hierarchical tasks 30
7.3 Dependencies between tasks 30
7.4 Modeling operating systems and hypervisors 30
7.5 Modeling power of software 30
Appendix A – Video Prototype Description 32

Revision History

Rev	Change	Date	Author
0.1	initial draft	6/3/17	dmista
0.2	added capabilities chapter, and clock/voltage and latency chapter	6/18/17	dmista
0.3	added details on clock/voltage trees and clock dividers	6/29/17	dmista
0.4	started on latency chapter	7/6/17	dmista
0.5	populated software and latency chapters	7/11/17	dmista
0.6	added bidirectional dependencies for control components	7/24/17	dmista
0.7	Joe's review	9/26/17	jmd
0.8	Davorin's review, removed chapter 8	10/3/17	dmista
1.0	Appendix A – Video prototype model added	11/1/17	dmista
1.1	Extended the introduction section	11/20/18	dmista

References

UHA Specification – Revision 1.0, 7/24/2017

13 About This Document

This purpose of this document is to explain how to create system level power models using the Unified Hardware Abstraction (UHA) modeling language.

This document aims to cover a range of different use cases.

It provides a basic introduction, using step-by-step refinement of a model and discusses specific topics in detail.

14 Modeling Basics

UHA descriptions consist of

- Models of hardware components and their power states
- Models of software and external events
- Dependencies between the software and the components

By modeling the power states of individual components as well as the dependencies between the components and the dependencies between software and the hardware, the power consumption of the overall system can be accurately estimated.

Before we create descriptions that represent specific systems, let's cover a few basics about UHA.

14.1 UHA Modeling Introduction

The main language elements of UHA are *objects* and *properties*. UHA objects describe objects such as components, software tasks, or power states. UHA properties describe individual attributes of an object.

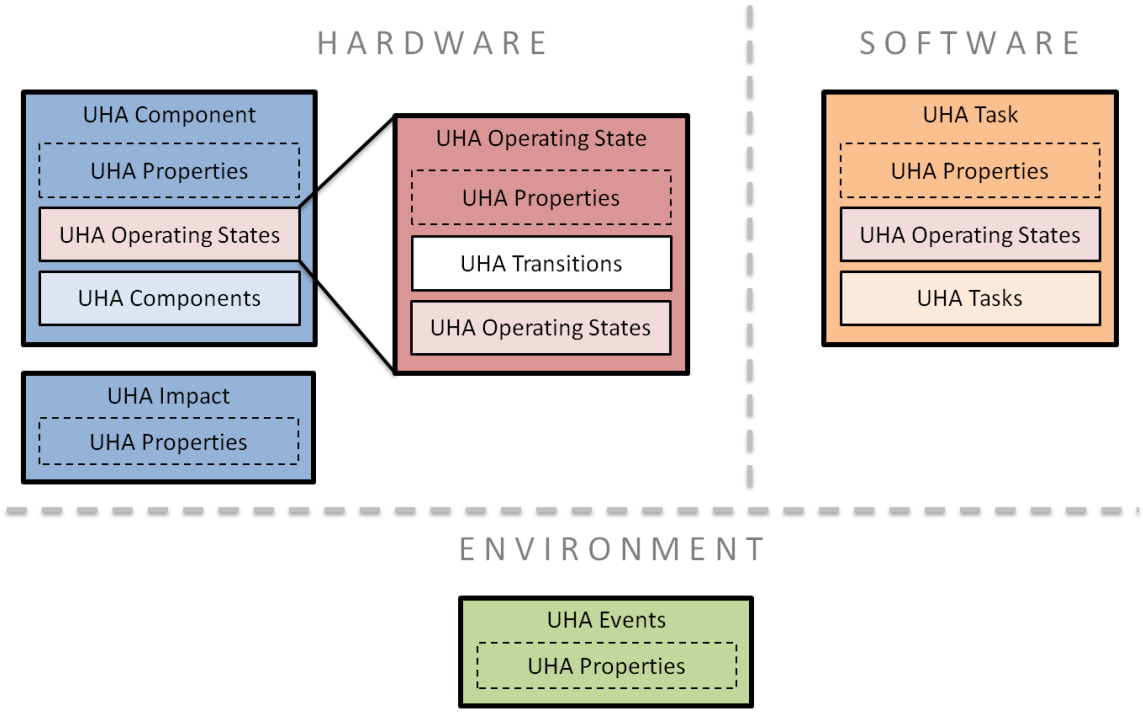


Figure 36 - Components of a UHA description

14.1.1 Modeling hardware components and power states

Hardware components are modeled as UHA objects. Hardware components can be hierarchical, i.e. a hardware component can have sub-components. A special type of UHA object is the power

state, or UHA state. Each component can have one or more states. States themselves can also be hierarchical.

Each component can have one or more power models as a UHA property. Power models can be specific to a state. Power models can either be expressed as equations or using lookup tables. Power models can reference properties defined anywhere in the model, i.e. component specific properties or system wide properties.

UHA transitions are used to describe the behavior of a component when a new state is entered. This includes the related energy profile, as well as the transition latency, i.e. the time it takes to perform the transition from the previous state to the new state.

UHA also allows modeling of so-called control components, i.e. clocks, voltage rails, or reset lines. Control components are an important detail for modeling the power states of digital systems as the majority of power states are controlled by either their clocks, their voltage rails, or their reset lines. Control components typically form hierarchies of their own, such as the clock tree or the voltage hierarchy.

14.1.2 Modeling software and external events

In UHA, software is modeled as UHA tasks, while the influence of hardware events is model as UHA impacts. UHA tasks and impacts may be hierarchical and they may have states (e.g. OS task is active or in a suspended state). Software states are an important factor in determining overall transition latencies as those latencies may be the gating factor for being able to enter low power states during periods of inactivity.

14.1.3 Modeling dependencies

Dependencies are a crucial aspect of UHA models ensuring that only valid combinations of states are being considered when simulating the energy related behavior of a system.

UHA dependencies for software tasks describe a software task's needs with respect to states of hardware components. UHA dependencies also extend to relationships between hardware components, specifically between the states of components.

Dependencies can be expressed explicitly using the "requires" property, referencing specific states of one or more components. More complex dependencies can be expressed as well, such as the requirement on a specific set of capabilities, or numeric requirements e.g. on a quality of service capability, or a certain memory throughput capacity. Dependencies on latencies can be expressed as well.

Control dependencies are established by identifying the respective control parent(s) of a component and then describing the required state of the control parents for a given state of the component. Control dependencies are especially important when one or more component shares the same clock or the same voltage rail, effectively linking the power states of several components despite those components potentially being functionally independent.

Control dependencies can also be referring to clock frequencies or voltage levels rather than just states.

14.2 UHA Syntax 101

This section contains a brief summary of the UHA language syntax; for more details please see the UHA reference manual.

14.2.1 UHA Objects

UHA Objects are defined using a name followed by braces (**{}**) containing the object body:

```
object-name { …body… }
```

The object body may contain properties followed by other objects, also referred to as *sub-objects*.

There are different types of UHA Objects, the standard type being a component. For all other types we use a type qualifier before the name, such as:

```
state state1 { …body… }
```

Here `state1` defines a **state**.

14.2.2 UHA Control Components

UHA allows explicitly modeling clocks, voltage rails, power islands, etc. by defining so called "control components."

```
voltage-source vdd1 { …body… }
clock clk1 { …body… }
reset vdd1 { …body… }
```

The code above shows how each of the three types of control components is defined.

14.2.3 UHA Properties

UHA Properties are defined using a simple assignment, i.e., a name and a value separated by an equal sign (=).

```
name = <property-value>;
```

Note that a semicolon (**;**) is mandatory following the property value.

There are different data types for properties, for a full list, see the UHA reference manual.

14.2.4 UHA Parameters

UHA Parameters are a type of property used to model properties that can change at runtime. Parameters are defined as follows:

```
name = parameter {
        [min = value;]
        [max = value;]
        [values = list_of_values;]
}
```

You can use the `min/max` fields to define a range or use the `values` field to define a list of permitted values.

14.3 Minimal UHA Description

Every UHA description needs to contain at least one root object, typically using the name of the system:

```
mysystem { …body… }
```

The `body` will contain the complete description of the system, its components and states, as well as its software and dependencies.

14.4 Managing larger descriptions

Descriptions can be distributed across multiple files, where the `include` directive can be used to concatenate multiple files.

If a description spans multiple files, each file needs to contain a root object, while the same name can be used for the root objects in every file. In fact, we recommend you use the same

root object name across all files of a single system to facilitate hierarchical descriptions and reuse.

15 A Simple System – Step-by-Step

This chapter shows how to describe a basic system with a single-core microcontroller and a few basic peripherals, starting with a single component and then refining it to incrementally add more detail.

15.1 Adding our first component

Let's start with a single component:

```
system {
   cpu {
         power = 1W;
   }
}
```

This description features a single component named `cpu` with a constant power consumption of `1 Watt`.

15.2 Adding states

In order to model a component that can change its state, we need to add UHA states to the model.

```
cpu {
   state active { power = 1W; }
   state inactive { power = 10mW; }
}
```

This description is for a CPU with active and inactive states.

15.3 Adding a dependency

If we add a second component, for example a memory, we can model our first dependency.

```
memory {
```

```
    state on { power = 2W; }
    state refresh { power = 1W; }
}
```

This description additionally defines a memory component with two states. We can now define a dependency as follows:

```
cpu {
    state active {
        requires = &memory/on;
            power = 1W;
        }
    state inactive { power = 10mW; }
}
```

This description declares a dependency of the cpu's `active` state on the memory's `on` state.

15.4 Adding power expressions

In order to model power more accurately, we can create power expressions incorporating properties and parameters.

```
    utilization = parameter {
        min = 0;
        max = 100;
    }
```

The code above defines a parameter specifying the utilization level of the CPU. The power model can reference this parameter by name.

```
    power = 500mW + utilization * 10mW;
```

Or, more elegantly:

```
power = base + utilization * coefficient;
```

where `base` and `coefficient` are defined as properties. The CPU model now looks as follows:

```
cpu {
      base = 500mW;
      coefficient = 10mW;
      utilization = parameter { min = 0; max = 100; }
      state active {
            requires = &memory/on;
            power = base + coefficient * utilization;
      }
      state inactive { power = 10mW; }
}
```

15.5 Adding frequency and voltage

UHA allows you to express dependencies on frequency and voltage by stating frequency and voltage levels as requirements. Furthermore, UHA allows referencing of frequency and voltage in power models.

```
cpu {
      …
      state active {
            requires = &memory/on, <frequency EQ 1GHz>;
            power = frequency * (…);
      }
      …
```

```
    }
```

When frequency and voltage requirements are expressed without defining any clocks or voltage sources explicitly, each component is assumed to have an ideal clock and voltage source which can provide whatever frequency or voltage level is required.

15.6 Adding an explicit clock

UHA allows modeling clocks, voltage rails, power islands, etc. explicitly by defining so called "control components."

```
clock clk1 { }
```

The component above defines a basic UHA clock. To connect the clock to a component, we can define a clock parent within the respective component.

```
cpu {
    clock-parent = &clk1;
    ...
    }
    memory {
    clock-parent = &clk1;
    ...
    }
```

In this description, we have both components sharing the same clock. As a result we now have a dependency between the `cpu` and the `memory`, as both will share the same clock frequency and clock gating will affect both components.

15.7 Adding an explicit voltage-supply

In the same way we defined explicit clocks, we can define a voltage-supply explicitly. A voltage-supply object can represent a voltage-rail or a power island.

```
voltage-supply vdd1 {
```

```
}
cpu {
      voltage-parent = &vdd1;
}
```

The code above describes a `cpu` that has `vdd1` as its voltage parent.

15.8 Modeling software tasks and events

UHA allows modeling of software by defining tasks. Tasks can specify requirements on hardware components.

```
task compute {
   requires = &cpu/active;
   begin = &start_computing;
   end = &end_computing;
   }
```

The task named `compute` above asserts the requirement for the `cpu` to be in its active state. The `begin` and `end` properties specify the UHA events that will trigger the start or end of the compute tasks.

```
event start_computing { }
event end_computing { }
```

The code above defines the events `start_computing` and `end_computing`.

16 Modeling Dependencies using Capabilities and Latencies

When a requirement of a task or component references a specific state of a component, we call that a *direct dependency*. Direct dependencies offer no flexibility as they represent a hard requirement. If another concurrent task requires a different state for the same component, then the resulting conflict cannot be resolved. In many cases, however, there is more than one state that will satisfy the needs of a task or component.

This kind of dependency can be expressed using capabilities as well as latencies.

16.1 Example: Memory with 3 states

Let's look at the example of a memory with three states:

- ON
- SR (aka SELF-REFRESH)
- OFF

If a task (e.g. a sleeping OS) requires the memory to preserve its contex,t we can specify a direct dependency as follows:

```
requires = &memory/SR;
```

This ensures the memory enters its self-refresh state. However, if another task or component required the memory to be in its `ON` state, then the two requirements cannot be satisfied simultaneously.

16.2 Defining capabilities

Let's extend the memory model by specifying capabilities for each of the states. We will introduce the following Boolean capabilities:

- `access`: the capability to access the memory via read/write operations
- `context`: the capability to preserve its context

We now assign the `access` capability to the `ON` state, while we assign the `context` capability to both the `ON` and the `SR` states.

```
memory {

   access = capability { }

   context = capability { }
```

```
   state ON { capabilities = access, context; }
   state SR { capabilities = context; }
   state OFF { }
}
```

16.3 Specifying capability requirements

Now, we can express the requirements of our tasks as follows:

```
   task sleeping_OS {
      requires = &memory/context;
   }
   task realtime_process {
      requires = &memory/access;
   }
```

The two tasks above can run concurrently because both requirements can be satisfied simultaneously. The ON state of the memory component satisfies both requirements. On the other hand, if the second task ends, then the description allows for the memory to enter its SR state which is likely to have a lower power consumption than the ON state, while still satisfying the "context" requirement of the first task.

16.4 Defining numeric capabilities

Capabilities that can be quantified with a number can define a numeric range.

```
   bandwidth = capability {
      min = 10;
      max = 100;
   }
   state ON {
      state max {
            capabilities = <bandwidth 100>;
```

```
    }
    state medium {
          capabilities = <bandwidth 50>;
    }
    state min {
          capabilities = <bandwidth 10;>;
    }
}
```

In the example above, we have a component with a bandwidth capability. The `ON` state has three sub-states `min`, `medium`, and `max` – each with their respective bandwidth capability. Obviously, the different sub-states will probably have their own frequency and voltage requirements and, therefore, differing power consumption and, potentially, also different power models.

16.5 Dependencies on numeric capabilities

Numeric capabilities such as the "bandwidth" defined above can be referenced in a requirement using the following expression:

```
requires = <&eth0/bandwidth GE 20>;
```

The code above states a dependency on the `eth0` component, where a bandwidth of `20` or more is required. Other possible operators are `EQ`, `GT`, `LT`, and `LE` to express equal, greater than, less than, and less than or equal to.

16.6 Latency requirements

UHA also allows components and tasks to assert requirements on latencies, either through state-latencies or capability-latencies.

```
requires = <&eth0/on/latency LT 10ms>;       // state
requires = <&memory/access/latency LT 500us>;      // capability
```

The actual latencies are defined on the state level, using entry and exit latency properties, or within a UHA transition.

```
eth0 {
   state on {
         entry-latency = 8ms;
}  }
memory {
   state on {
         transition {      // transition from state "off"
               from = &off;
               latency = 12ms;
}  }     }
```

For more on modeling latencies, see Chapter 18.

17 Modeling Voltage and Clock Trees

Clocks and voltages are an important factor when modeling the power of a system, both in terms of the power models, as well as related to power state transitions. UHA allows you to cover both aspects, either by modeling just voltage and frequency levels per component or by explicitly modeling clocks and voltage components.

17.1 Modeling frequency and voltage per component

Using UHA, it is possible to model voltage and frequency without defining any clocks or voltage supplies explicitly. In this case, each component is assumed to have an ideal voltage source and an ideal clock source. This allows modeling of frequency and voltage impacts on the power consumption of individual components, while ignoring any interdependencies that may arise from shared clocks or common voltage rails. This type of modeling may be desirable at the early stage of modeling.

For a component to specify a frequency or voltage level, it needs to assert a requirement accordingly.

```
cpu {
      state active {
            requires = <frequency EQ 1GHz>, <voltage EQ 1.1V>;
            power = frequency * (…) + voltage * (…);
      }
}
```

The `cpu` modeled above will set its frequency to `1GHz` and voltage to `1.1 Volt`.

17.2 Modeling clocks and voltage components explicitly

To capture the actual hierarchies for clock, voltage, and reset, UHA allows the explicit definition of control components as introduced in Section 14.2.2.

In addition to asserting frequency and voltage requirements, each component may also specify control parents, i.e., clock-parent or voltage-parent, in order to identify the clock and voltage component that controls the voltage level and clock frequency.

```
cpu {
      clock-parent = &c1;
      voltage-parent = &v1;
      state active {
            requires = <frequency EQ 1GHz>, <voltage EQ 1.1V>;
            power = frequency * (…) + voltage * (…);
      }
}
```

If more than one component specifies the same control-parent, it is implied both components will be subject to the same clock-frequency or voltage level. This also implies both components would be affected by state changes of that control component, e.g., if a shared clock is gated, all of the components it supplies would have their clock gated.

17.2.1 Components with multiple clocks or voltage rails

UHA permits only one control parent of each type. I.e., each component can have only one clock parent, one voltage parent, and one reset parent.

A component that has more than one clock or more than one voltage rail will need to be modeled using individual sub-components, so each sub-component has exactly one control parent of each type.

17.2.2 Modeling clock gates and power gates

If two components are subject to the same clock frequency, but can be gated individually, then it is possible to model this by creating a two-level clock hierarchy, where the top-level clock controls the frequency, while the lower-level clocks only permit gating.

```
clock clk_top {
    frequency-output = capability {
        min = 100MHz;
        max = 1GHz;
    }
}
clock clk1 {
```

```
        clock-parent = &clk_top;
    }
    clock clk2 {
        clock-parent = &clk_top;
    }
```

By default, each clock and each voltage-supply component has two states, `ON` and `OFF`. Also by default, a clock inherits its frequency from its parent. Hence, the two clocks `clk1` and `clk2` are clocks that can be gated and they share the same frequency as provided by `clk_top`.

If described explicitly, `clk1` would look like:

```
    clock clk1 {
        clock-parent = &clk_top;
        state ON { requires = clock-parent/ON; }
        state OFF { }
    }
```

In the example above, the `ON` state of `clk1` requires `clk_top` to also be *ON*. However, if `clk_top` enters its `ON` state due to other dependencies, `clk1` will not automatically also transition to its `ON` state.

17.2.3 Modeling clocks without clock-gating or power-gating capability

To describe a clock hierarchy where one of the clocks does not allow gating, you can describe a bi-directional dependency to its parent's states using the `clock-state` property.

```
    clock clk_nogate {
        clock-parent = &clk_top;
        state ON { clock-state = ON; }
        state OFF { clock-state = OFF; }
    }
```

The description above basically states that whenever `clk_top` changes its state, `clk_nogate` will change states accordingly.

For voltage related dependencies, use the equivalent `voltage-state` property to express a bi-directional dependency.

17.2.4 Modeling clock dividers

The `values` field of a capability can be defined using an expression rather than a list. This can be used to express the relationship between the input frequency (i.e., the frequency provided by the clock parent) and the output frequency.

```
clock clk_div {
    clock-parent = &clk_top;
    frequency-output = capability {
        values = frequency / 2;
}
```

The example above shows how to model a fixed clock divider; in this case, the output clock of `clk_div` is ½ of its input clock.

Parameters can be used to model a variable divider:

```
clock clk_var {
    clock-parent = &clk_top;
    divider = parameter {
        values = 1, 2, 4, 8, 16;
    }
    frequency-output = capability {
        values = frequency / divider;
    }
}
```

The example above describes a clock with a variable divider, supporting a set of discrete values from `1` to `16`.

18 Modeling Latencies

Modeling latencies in UHA serves two purposes. On one hand, latencies capture the delays incurred when state transitions are performed, which allows tools to report and analyze such latencies. On the other hand, UHA also allows you to express dependencies in terms of latency requirements.

Latencies are modeled in UHA by defining state transition latencies for each component in the hierarchy. The transition latencies captured for one component shall only account for the time it takes for the component to transition from the previous state to the new state -- they should not include the latency of other state changes triggered by any of the components' requirements, such as changing of a clock frequency or turning on a clock or voltage source.

The total latency for a components' state transition can be computed by aggregating the latency of the components' state transition with the latencies of all state transitions resulting from the dependencies.

18.1 Defining entry and exit latency properties

A UHA state may define entry and exit latency properties. The *transition latency* is computed by adding the exit-latency of the old state or source state and the entry-latency of the new state or target state.

```
eth0 {
   state on {
         entry-latency = 8ms;
         }
         state idle { }
         state off {
               exit-latency = 5ms;
         }
}
```

The code above describes a component with its transition latencies. The total latency for entering the `on` state is either `8ms` when transitioning from the `idle` state or `13ms` when transitioning from the `off` state.

18.2 Defining latencies within a transition

When transitions are defined explicitly, they may also contain a `latency` property. This property explicitly specifies the transition latency between pairs of states, i.e., the source and target state.

```
memory {
   state on {
         transition {      // transition from any state
               latency = 3ms;
         }
         transition {      // transition from state "off"
               from = &off;
               latency = 12ms;
}  }     }
```

In the example above, because the transition from source state `off` is defined explicitly, it supersedes the generic transition specified without a `from` property. Hence, the transition latency into the `on` state is either `12ms` from the `off` state or it is `3ms` (for all other cases).

18.3 Mixing explicit transition latencies and entry/exit latencies

Entry and exit latencies of the target and source state only apply if no explicit transition latency for the current combination of source and target states is defined. Even if the transition latency is defined without a reference to a specific source state it supersedes any latency specified using the entry- or exit-latency properties.

```
memory {
   state on {
         entry-latency = 5ms;
         transition {
               latency = 3ms;
   }     }
```

```
    state idle {
          exit-latency = 1ms;
}   }
```

In the example above, the latency for transitioning into the `on` state is always `3ms`, regardless of the source state. In other words, the entry and exit latencies specified in the example above are ignored whenever transitioning into the `on` state.

19 Modeling Software

Software is modeled using UHA tasks. A UHA task may represent a sub-routine, an entire application, or an operating system. It is up to you to choose the appropriate level of detailed required, depending on the modeling objectives and the complexity of the system.

19.1 Differences between software and hardware components

The main difference between UHA tasks and UHA components is that hardware components are assumed to always be present while software isn't, i.e., it must first be loaded or started. As components are always present, they will always have a state as soon as the system starts. Tasks, on the other hand, are assumed to be not loaded once the system starts; instead, they need to be started explicitly, usually through a scenario file or stimulus in the simulator.

Tasks may have states as well, but aren't required to. Application tasks that just run until they complete do not need to model any states. Background and supporting tasks or services, however, may include states in order to reflect the fact they're loaded, but idle, versus actively processing data.

19.2 Hierarchical tasks

Tasks can have children, also referred to as subtasks. Subtasks can only run if their parent is running as well.

19.3 Dependencies between tasks

The requirements specified in a task may relate to other tasks being loaded or in a particular state.

19.4 Modeling operating systems and hypervisors

Software, such as operating systems or hypervisors, can be modeled using UHA tasks as well. Typically, such tasks would also include states, to account for the fact an OS can be in a sleep state, and that there's a significant latency involved when transitioning into or out of a sleep state, e.g., due to context saving.

A task dependency can be used to model an operating system being in a sleep state until a processing task starts, which requires the OS task to be in its active state.

19.5 Modeling power of software

In real systems, software doesn't consume power, instead it triggers state changes in the hardware resulting in certain power profiles consumed by the hardware components. In an ideal UHA model, the complete power consumption of any software task is captured by the power models across the hardware components based on the hardware requirements of the

UHA tasks. However, in practice, the energy profile of the hardware components will not always match the power consumption you could measure on a real device.

Discrepancies between the measured power and the modeled power can be compensated for using an additional power model dedicated to the task in question.

Appendix A – Video Prototype Description

The following source code represents the complete description of the video prototype as specified in the system specification contributed to IEEE P2415 by Broadcom.

```
video_prototype {
  description = "Video Prototype System";

  cpu {
    voltage_parent = &pmic/sr1;
    clock_parent = &pll2;

    PrcActivePowerWFI = 10;
    mAPerMHz = 300uA;
    leakage = 10mA;

    temp = parameter {
        default = 40;
    }

    state ACTIVE {
        clock_state = ON;
        power = voltage * ((mAPerMHz / 1000000) * frequency * voltage
                + leakage * 2 ^ ((temp - 25) / 20));

        state max {
            requires = <frequency EQ 1GHz>;
            requires = <voltage EQ 1200mV>;
        }
```

```
        state lowpower {
            requires = <frequency EQ 333MHz>;
            requires = <voltage EQ 800mV>;
        }
    }
    state INACTIVE {
        state wfi {
            clock_state = ON;
            requires = <frequency EQ 333MHz>;
            requires = <voltage EQ 800mV>;
            power = voltage * ((mAPerMHz / 1000000) * frequency
      * (PrcActivePowerWFI / 100)
            + leakage * 2 ^ ((temp - 25) / 20));
        }
        state off {
            voltage_state = OFF;
            clock_state = OFF;
            power = 0;
        }
    }
}

memc {
    voltage_parent = &pmic/sr2;
    clock_parent = &clk_div22;

    temp = parameter {
    default = 40;
    }
```

```
bandwidth = parameter {
default = 0;
}

mAPerMBps = 100uA;
leakage = 5mA;
mAGated = 5mA;

state ACTIVE {
    state on {
        requires = <voltage GE 900mV>;
        requires = <frequency GE 400MHz>;
        power = voltage * (mAPerMBps * (voltage / 1V) *
        (frequency / 400MHz) * bandwidth + leakage *
        2 ^ ((temp - 25) / 20));
    }
}
state INACTIVE {
    state gated {
        requires = <voltage GE 900mV>;
        requires = <frequency GE 400MHz>;
        power = voltage * (mAGated * (voltage / 1V) *
        (frequency / 400MHz) + leakage * 2 ^ ((temp - 25) / 20));
    }
    state off {
        voltage_state = OFF;
        power = 0;
    }
}
```

```
}

voltage_supply power_switch {
    voltage_parent = &pmic/sr2;

    state ON {
        requires = &voltage_parent/ON;
        power = 0;
    }
    state OFF {
        power = 0;
    }
}

video_hw {
    voltage_parent = &power_switch;
    clock_parent = &clk_div21;

    temp = parameter {
    default = 40;
    }
    mAEncode = 100mA;
    mADecode = 50mA;
    mADisplay = 25mA;
    leakage = 5mA;

    state ACTIVE {
        requires = <voltage GE 900mV>;
        requires = <frequency GE 500MHz>;
```

```
        state encode { power = voltage * (mAEncode * (voltage / 1V)
    *(frequency / 500MHz) + leakage * 2^((temp - 25) / 20)); }
        state decode { power = voltage * (mADecode * (voltage / 1V)
    *(frequency / 500MHz) + leakage * 2^((temp - 25) / 20)); }
        state display { power = voltage * (mADisplay * (voltage / 1V)
          *(frequency / 500MHz) + leakage * 2^((temp - 25) / 20)); }
    }
    state INACTIVE {
        state gated {
            requires = <voltage GE 900mV>;
            requires = <frequency GE 500MHz>;
            power = voltage * leakage * 2 ^ ((temp - 25) / 20);
        }
        state off {
            voltage_state = OFF;
            clock_state = OFF;
            power = 0;
        }
    }
}

dram {
    voltage_parent = &pmic/sr3;

    /* Runtime parameters */
    temp = parameter {
    default = 35;
    }
```

```
RDprc = parameter {
default = 0;
}
WRprc = parameter {
default = 0;
}

/* Implementation and design characterisics */
/* In GB */
DRAM_SIZE = 1;
Ivdd2_SR = 850uA;
IvddQ_SR = 20uA;
Ivdd2_RW = 200mA;
IvddQ_RW = 200mA;
Ivdd2_Idle = 20mA;
IvddQ_Idle = 1mA;

state ACTIVE {
    requires = <voltage EQ 1.2>, &dram_io/ACTIVE;
    power = voltage * (((RDprc + WRprc) / 100) * (Ivdd2_RW + IvddQ_RW)
+ (1 - (RDprc + WRprc) / 100) * (Ivdd2_Idle + IvddQ_Idle));
}
state SELF_REFRESH {
    requires = <voltage EQ 1.2>, &dram_io/SELF_REFRESH;
    power = voltage * (Ivdd2_SR + IvddQ_SR) * DRAM_SIZE *
      2 ^ ((temp - 25) / 20);
}

dram_io {
    voltage_parent = &pmic/sr4;
```

```
        /* Implementation and design characterisics */
        Ivdd1_SR = 400uA;
        Ivdd1_RW = 5mA;
        Ivdd1_Idle = 3mA;

        state ACTIVE {
            requires = <voltage EQ 1.8V>;
            power = voltage * (((&dram/RDprc + &dram/WRprc) / 100)
                * Ivdd1_RW + (1 - (&dram/RDprc + &dram/WRprc) / 100)
                * Ivdd1_Idle);
        }
        state SELF_REFRESH {
            requires = <voltage EQ 1.8V>;
            power = voltage * Ivdd1_SR * &dram/DRAM_SIZE *
                2 ^ ((&dram/temp - 25) / 20);
        }
    }
}

events {
    event video_start {}
    event video_end {}
    event video_decode_start {}
    event video_decode_end {}
    event video_display_start {}
    event video_display_end {}
    event video_idle_start {}
    event video_idle_end {}
```

```
}

tasks {
    task video_60fps {
        fps = parameter {
        default = 60; // quality of the stream
        }
        begin = &video_start;
        end = &video_end;

        task video_decode {
            begin = &video_decode_start;
            end = &video_decode_end, &video_display_start;

            requires = &dram/ACTIVE, &cpu/ACTIVE/max,
                 &memc/ACTIVE/on, &video_hw/ACTIVE/decode, &dig/ON;

            set = <&memc/bandwidth 1200>;
            set = <&dram/RDprc 13.75>;
            set = <&dram/WRprc 5>;
        }

        task video_display {
            begin = &video_display_start;
            end = &video_display_end, &video_idle_start;

            requires = &dram/ACTIVE, &cpu/ACTIVE/lowpower,
           &memc/ACTIVE/on, &video_hw/ACTIVE/display, &dig/ON;
```

```
            set = <&memc/bandwidth 800>;
            set = <&dram/RDprc 9.5>;
            set = <&dram/WRprc 3>;
        }

        task video_idle {
            begin = &video_idle_start;
            end = &video_idle_end, video_decode_start;

            requires = &dram/SELF_REFRESH, &cpu/INACTIVE/wfi,
    &memc/INACTIVE/gated, &video_hw/INACTIVE/gated, &dig/ON;

            set = <&memc/bandwidth 0>;
            set = <&dram/RDprc 0>;
            set = <&dram/WRprc 0>;
        }
    }
 }
}
```