CHALMERS

Evaluation of Digital Power Control Algorithms for Automotive LED Headlights using TMS320F28035 Microcontroller

Master of Science Thesis in Integrated Electronic System Design

MUHAMMAD WAQAR AZHAR

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, July 2011

Evaluation of Digital Power Control Algorithms for Automotive LED Headlights using TMS320F28035 Microcontroller

Muhammad Waqar Azhar

Examiner: Prof. Per Larsson-Edefors

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, July 2011

ACKNOWLEDGEMENTS

First of all. thanks to ALLAH Almighty for enabling me to complete this thesis and all the blessings upon me. Special thanks to Ole-Kristian Skroppa for supervising my thesis and the support and motivation that he gave throughout this project. I would like to extend my gratitude towards my internal supervisor Professor Per Larsson-Edefors and for his valuable feedback throughout my master degree. Moreover, I would like to thank the automotive application team, especially Christoph Mendth, Matthias Terhorst, Roberto Scibilia and Josef Mieslinger for their valuable help during different phases of project.
I would like to dedicate this work to my parents, my intelligent and hardworking father who has been an inspiration all through my life and my mother who's love and care has enabled me to reach this far in professional life. Last but not the least i would like to thank rest of my family and friends for their extended support

Abstract

Digital power management solutions offer some distinct advantages over their analog counterparts and are increasingly being employed by power electronics designers these days. The post-implementation flexibility of digital solutions is one of the major advantages. Increasing demands on functionalities like communication, remote fault monitoring and configurability have also fueled this transition from analog to digital control. The possibility of implementing non-linear control algorithms is a major advantage of these solutions. Highly integrated analog components, like analog to digital converters and comparators in modern microcontrollers, have enabled designers to decrease on-board component count and complexity. Digital power management solutions have limitations in performance when compared to analog designs, but rapid performance improvements in microcontrollers (MCUs) certainly create a bright future for digital power management.

In this project we have investigated a digital power control implementation for switch mode DC-DC converters. The application under consideration is LED-based automotive headlights. LEDs (light emitting diodes) have gained foothold in many lighting applications due to the decrease in cost per lumen. The automotive industry has previously employed LEDs in a number of lighting application, like back and interior lighting, and now consider a potential use in headlights. Moreover we wanted to prove the capabilities of the TMS320F28035 MCU that is designed for real-time control applications. The combination of two new avenues of digital power management and LED headlights has raised a few challenges that have been solved in this project.

First, the characteristic behavior of LEDs is different from conventional incandescent bulbs. LEDs are controlled by maintaining a constant current through them rather than applying a constant voltage. Switch mode power stages intended to control LEDs inherently operate in voltage mode. Considerable modeling and implementation efforts are required to handle both these contrasting behaviors. Secondly, discrete-time models are required for MCU implementation. Existing control theory predominantly employs continuous-time models. These continuous-time models are required to be converted to discrete-time models to make use of existing models for digital implementation. Discrete-time models that are developed from scratch requires considerable effort. Third, existing test-setups used for analog designs can not be directly used for digital designs. Considerable analysis is required for these modifications.

The first step in the design flow is modeling; we have to model converters for controlling current through LEDs. Two solutions are proposed: First, existing continuous-time voltage mode models for converters are used and modified to control LEDs. Later some further modifications are made to convert them to discrete time. Second, a new non-linear and discrete-time model is proposed for controlling LED current using inductor current feedback.

Later, the developed model is implemented on TMS320F28035. This MCU contains two heterogeneous processor cores. A pre-implementation analysis is carried out to allocate hardware resources and system bandwidth to different software tasks. The control loop is implemented on a so-called control-law accelerator, as it is optimized for this purpose, while useful features like graphical user interface and diagnostics are implemented on C28x CPU. The initial bandwidth allocation to different software tasks is verified by doing measurements and re-allocation. The initial implementation of different software components is also optimized to enhance performance based on this post-implementation system analysis.

Lastly, the implemented control algorithms are verified by performing frequency response measurements. Modifications are made to existing test-setups to suit them to needs of digital power control. Open loop and closed loop measurements are performed under different operating conditions. These measurements are compared with results from the model and used to lay down our final analysis. Recursive design approach is used at each design phase. Moreover previous design phases are revisited whenever necessary to optimize the implementation.

Contents

1 Background and Theory **9**
- 1.1 Problem Statement . . . 9
- 1.2 Project Work Flow . . . 10
- 1.3 What is LED . . . 10
- 1.4 Application of LEDs in Automotive Lighting . . . 11
- 1.5 LEDs Characteristic Behavior . . . 13
- 1.6 Comparison Between Linear and Switch Mode Control . . . 14
- 1.7 Switch Mode DC/DC Power Converter . . . 16
  - 1.7.1 Buck Converter . . . 16
  - 1.7.2 Boost Converter . . . 16
  - 1.7.3 SEPIC Converter . . . 17
  - 1.7.4 Comparison Between Different SMPS Topologies . . . 18
- 1.8 Comparison Between Analog and Digital Power Management . . . 19
  - 1.8.1 Analog Controller . . . 20
  - 1.8.2 Digital Power Management . . . 20
  - 1.8.3 Continuous vs Discrete Time Compensation . . . 21
  - 1.8.4 Comparison between Analog and Digital Power Management 21

2 Hardware Platform **23**
- 2.1 Evaluation Module Overview . . . 23
- 2.2 SMPS stages . . . 24
  - 2.2.1 Buck SMPS Stage . . . 24
  - 2.2.2 Boost SMPS Stage . . . 24
  - 2.2.3 Buck-Boost SMPS Stage . . . 24
  - 2.2.4 SEPIC SMPS Stage . . . 25
- 2.3 Central Controller . . . 26
  - 2.3.1 TMS320F28035 Real-Time Micro-Controller . . . 26
  - 2.3.2 Control Law Accelerator . . . 28
  - 2.3.3 Analog to Digital Converter . . . 30
  - 2.3.4 Enhanced Pulse Width Modulation Module . . . 31
- 2.4 Pin Assignments . . . 33

3 Design Considerations for LED Automotive Headlights 35
3.1 Design Matrices . . . 35
3.1.1 SMPS Modeling . . . 35
3.1.2 Switching Frequency . . . 36
3.1.3 Current Control vs Voltage Control . . . 37
3.1.4 Life Cycle and Diagnostics . . . 37

4 Modeling and Compensation 38
4.1 Voltage-Mode Control . . . 38
4.1.1 Modeling of SMPS . . . 39
4.1.2 Compensation . . . 45
4.2 Current-Mode Control . . . 47

5 System Architecture and Firmware 52
5.1 Software Development Tools and Resources . . . 52
5.1.1 Digital Power Library . . . 52
5.2 System Overview . . . 52
5.3 Challenges Related to Discrete System Implementation . . . 54
5.4 Hardware Initialization . . . 55
5.4.1 PWM Initialization . . . 56
5.4.2 ADC Initialization . . . 57
5.4.3 CLA Initialization . . . 58
5.5 Control Loop Implementation . . . 58
5.5.1 Control Loop Architecture . . . 59
5.5.2 System Architecture . . . 60
5.5.3 Improvements in Control Loop Architecture . . . 62
5.5.4 Control Loop Triggering . . . 64
5.6 C28x Code . . . 65
5.7 PWM Dimming . . . 69
5.7.1 Limitation I . . . 70
5.7.2 Limitation II . . . 71
5.8 Diagnostics . . . 73
5.8.1 Over-Voltage Protection . . . 74

6 Measurement and Results 78
6.1 Measurement Equipment . . . 78
6.2 Open-Loop Measurements . . . 79
6.2.1 Method I . . . 80
6.2.2 Method II . . . 80
6.2.3 Method III . . . 81
6.2.4 Digital Controller Measurement Setup . . . 81

6.2.5 Measurement Results 84
6.3 Closed-Loop Measurement 86
6.4 Hardware Platform Improvement 90

7 Conclusion **92**

A Matlab Models **98**
A.1 Buck 98
A.2 Boost 98
A.3 Buck-Boost 99
A.4 SEPIC 100
A.5 Filter Conversion Script 100
A.5.1 Convert.m 100
A.5.2 Convert2.m 101
A.6 Current Mode Control 101

B Firmware **102**
B.1 Main.c 102
B.2 2nd Order IIR Filter code 117
B.3 3rd Order IIR Filter code 118
B.4 CLA Code 119
B.5 CLA ISR used for Dimming 121
B.6 PWM Initialization 122

C Source Code for Open Loop Measurement for Buck SMPS **124**
C.1 CLA Code 124
C.2 Modifications in Main Code 125

List of Figures

1.1 Comparison of Luminous Efficiency based on Haitz Law 11
1.2 Future trends in LED device efficiency 13
1.3 Characteristic of an Ultra White Automotive LED [1] 14
1.4 Linearly controlled LED setup [2] 15
1.5 Generic buck architecture. 17
1.6 Generic boost architecture. 18
1.7 Generic SEPIC architecture. 19
1.8 Block diagram of SMPS controlled by analog controller 20
1.9 Block diagram of SMPS controlled by digital controller 21

2.1 General EVM board layout 24
2.2 Detailed implementation of buck SMPS stage 25
2.4 Detailed implementation of buck-boost SMPS stage 25
2.3 Detailed implementation of boost SMPS stage 26
2.5 Detailed implementation of SEPIC SMPS stage 26
2.6 Functional block diagram of TM320F28035 [3] 27

4.1 Voltage control mode 39
4.2 Bode plot of Matlab model for buck converter 42
4.3 Bode plot of Matlab model for boost converter 44
4.4 Bode plot of Matlab model for buck-boost converter 45
4.5 Bode plot of Matlab model for SEPIC converter 46
4.6 Bode plot of compensated Matlab model for buck converter 47
4.7 Bode plot of compensated Matlab model for boost converter . . . 48
4.8 Two switching states of boost converter [4] 49
4.9 Bode plot for current mode control model for boost converter . . . 51

5.1 General software architecture. 53
5.2 Sampling true average current 54
5.3 Symmetric PWM waveform 56
5.4 Synchronization of PWM and triggering of ADC channels 57
5.5 CLA task control flow diagram 58

5.6 3-pole 3-zero filter interfacing in control loop 60
5.7 High resolution PWM driver 61
5.8 Control loop system implementation 62
5.9 Modified control loop system implementation 63
5.10 Computation sequence of control loop 64
5.11 Time line diagram of four control loops running on CLA 65
5.12 C28x software architecture overview 66
5.13 State Machine A 66
5.14 State Machine B 67
5.15 State Machine C 68
5.16 Enable dimming task architecture 69
5.17 Circuit Description of Dimming 70
5.18 Triggering of ISR handling dimming function 71
5.19 Screen shot of dimming 71
5.20 Screen shot of dimming after improvements 73
5.21 Over-voltage protection implementation I 75
5.22 Over-voltage protection implementation II 76
5.23 OVP screen shot for boost SMPS 77

6.1 Open-loop measurement circuit setup I [5] 80
6.2 Open-loop measurement circuit setup II [5] 80
6.3 Open-loop measurement circuit setup III [5] 81
6.4 Open loop gain for buck converter 81
6.5 Open-loop measurement setup 83
6.6 Open-loop gain for buck converter 84
6.7 Open-loop gain for boost converter 85
6.8 Open-loop gain for SEPIC converter 85
6.9 Classical method for measuring frequency response [5] 86
6.10 Venable method for measuring frequency response [5] 87
6.11 Closed-loop measurement setup 88
6.12 Boost feedback loop measurement 88
6.13 Buck feedback loop measurement setup 89

List of Tables

1.1 Comparison between conventional and LED light efficiency [6] . . 12
1.2 Comparison of analog and digital controller performance [7] . . . 22

2.1 Detailed list of peripherals in TMS320F28035 MCU [8] 28
2.2 Pin connection on EVM 33

List of Abbreviations

- ADC-Analog to Digital Converter
- CPU-Central Processing Unit
- DC-Direct Current
- ESR-Equivalent Series Resistance
- EVM-Evaluation Module
- GUI-Generic User Interface
- ISR-Interrupt Service Routine
- GPIO-General Purpose Input Output
- JTAG-Joint Task Action Group
- MCU-Micro Controller Unit
- FET-Field Effect Transistors
- OPAmp-Operational Amplifier
- PCB-Printed Circuit Board
- PWM-Pulse Width Modulation
- ePWM-Enhanced Pulse Width Modulation
- RISC-Reduced Instruction Set
- SMPS-Switch Mode Power Supplies
- TI-Texas Instruments
- USB-Universal Serial Bus

- DSP-Digital Signal Processor
- CLA-Control Law Accelerator
- HRPWM-High Resolution Pulse Width Modulation
- SYSCLKOUT-System Clock Signal
- RAM-Random Access Memory
- LED-Light Emitting Diode
- SAR-Successive approximation Register
- SOC-ADC channel control logic
- DPL-Digital Power Library
- IIR-Infinite Impulse Responce
- CCS-Code Composer Studio
- IDE-Integrated Development Environment

Chapter 1

Background and Theory

1.1 Problem Statement

Digital control is being employed by commercial power supply designers. This comprises both digital circuits and micro-controller (MCU) based solutions. Digital circuit solutions give better performance and optimization because they are tailored to specifications, while MCU based solutions offer considerable performance with low cost and less time to market. Digital power control offers significant advantages over analog control.

The purpose of this project is to evaluate digital power control for switch mode power supplies (SMPS) driving LEDs. LED based automotive headlights is the underlined product. This project deals with two combined problems. First, modeling of power LEDs and designing a control strategy for them using switch mode power supplies. Secondly, using digital power control for this implementation. The major design tasks are depicted below

- Efficient control algorithms for LEDs headlamps
- Optimal algorithm for pulse width modulation (PWM) dimming of LED headlamps
- Models for current mode control of switch mode power supplies
- Compensation based on above model
- MCU implementation and its trade-offs
- The relative importance of different diagnostic function has to be determined to have an optimal implementation, as total system bandwidth is limited

Modeling of SMPS in current mode needs considerable design space exploration because of lack of available published work. Different implementation strategies are to be analyzed and solved, furthermore a post-implementation analysis of system behavior and performance is also desirable to get a measure of effectiveness.

1.2 Project Work Flow

First, SMPS driving power LEDs are modeled in MATLAB to evaluate system behavior. Later on these models are used for MCU based implementation. The implementation consists of control law, communication and diagnostic functions. Texas Instrument's TMS320F28035 real-time micro-controller is used for implementation of digital power control algorithms. Later, performance of these implemented algorithms will be evaluated using a frequency response analyzer. A recursive design approach between different design phases is employed to search optimal algorithms and implementation.

1.3 What is LED

LEDs differ from traditional light sources. In an incandescent lamp, a tungsten filament is heated by electric current until it glows or emits light. In a fluorescent lamp, an electric arc excites mercury atoms, which emit ultraviolet (UV) radiation. After striking the phosphor coating on the inside of glass tubes, the UV radiation is converted and emitted as visible light. A LED, in contrast, is a semiconductor diode. It consists of a semiconductor p-n junction. When properly biased, current flows from the p-side (anode) to the n-side (cathode) and light is emitted [9]. However, silicon is unsuitable for making LEDs because the so-called energy band gap is too low. The first LEDs were made from gallium arsenide (GaAs) and produced infrared light at about 905 nm. The reason for producing this color is the energy band gap, that is, the difference between the conduction band and the lowest energy level (valence band) in GaAs. When a voltage is applied across the LED, electrons are given enough energy to jump into the conduction band and current flows. When an electron loses energy and falls back into the low energy state (the valence band), a photon (light) is often emitted [10].

LEDs provide several advantages over tradition light sources; as summarized below

- Long life
- Robustness

- Small size
- Non-toxicity
- Versatility
- High energy efficiency

1.4 Application of LEDs in Automotive Lighting

With advances in semiconductor technology, LEDs have started getting attention for usage as light sources. Recent advancements have considerably decreased the cost of Lumen per Watt to such a level that LEDs have started finding foothold in a lot of applications. Haitz law—a LED counterpart of Moore's law for integrated circuits—states that every decade, the cost per lumen falls by a factor of 10 and the amount of light generated per LED package increases by a factor of 20, for a given wavelength of light. A comparison between projected data based on Haitz law and real data for LED efficiency for past decade is shown in figure 1.1 [11].

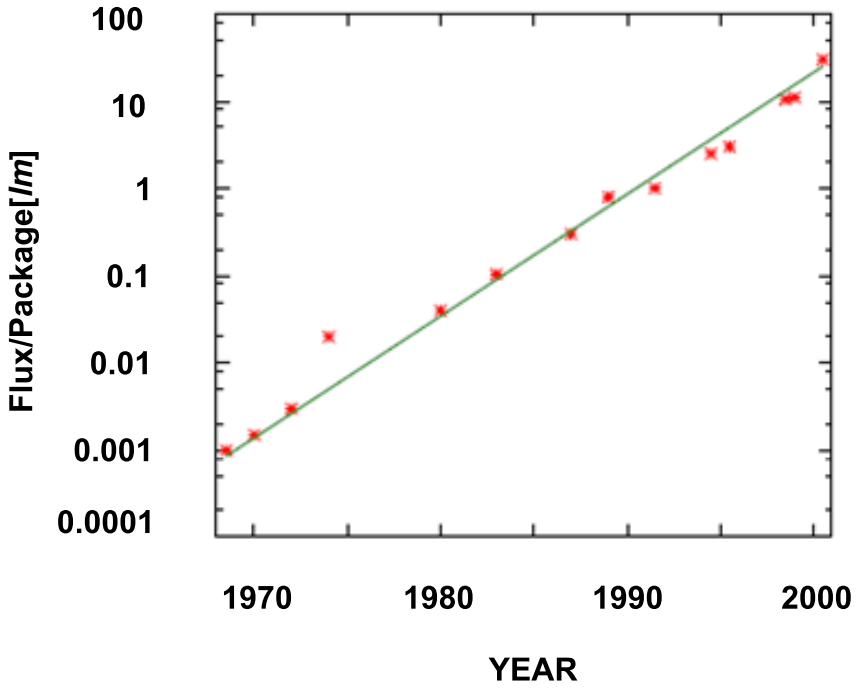

Figure 1.1: Comparison of Luminous Efficiency based on Haitz Law

The efficiency of LEDs has increased over the last decade. A lot of research has been done and power LEDs manufacturers are trying to keep up with the latest developments. Before looking into the future trends in LEDs it is worthwhile to put down a comparison presented in table 1.4, conducted by United States Department of Energy [6].

	CFL	LED
Light Source		
Lamp lumen rating (lm)	860	
Light source wattage (W)	13	1
LED manufacturer declared "typical luminous flux" (lm/led)		100
Number of lamps/LEDs per fixture	1	12
Luminaire measurements		
Luminaire lumens (lm)	514	589
Measured luminaire wattage (W)	12	14
Fixture efficiency	60	
Luminaire efficacy (lm/W)	42	42

Table 1.1: Comparison between conventional and LED light efficiency [6]

LEDs have gained a significant ground in recent years and are predicted to outperform traditional light sources on the parameters of cost and efficiency. A study conducted by US Department of Energy (see figure 1.2) presents the past development and future trends of commercial and laboratory based LED devices [12].

LEDs are experiencing an exponential growth, like integrated circuits density have experienced in last decade. LEDs are becoming increasingly efficient, making them likely and natural successors to conventional lighting solutions. Concerning automotive headlights, this replacement has started with some premium cars having LED headlights. A continuous decrease in cost of Lumen per Watt will certainly enable manufactures to use LED-based headlights in mid range cars in the coming years.

High efficiency is also a major factor. There has been a market shift toward energy-efficient cars in recent years such as hybrid cars. LED-based lighting solutions, being more energy efficient, are more suitable for such cars. According to a study, LED-based headlights can save 40% energy compared to halogen lamp [13]. LED-based headlights are more compact [14]. LEDs currently dominate the tail and center-mounted brake light markets. The use of LED in headlamps is expected to reach 26% of the market by 2015, mainly replacing halogen lamps [15]. Moreover, LED-based lighting solutions have a significant safety advantage

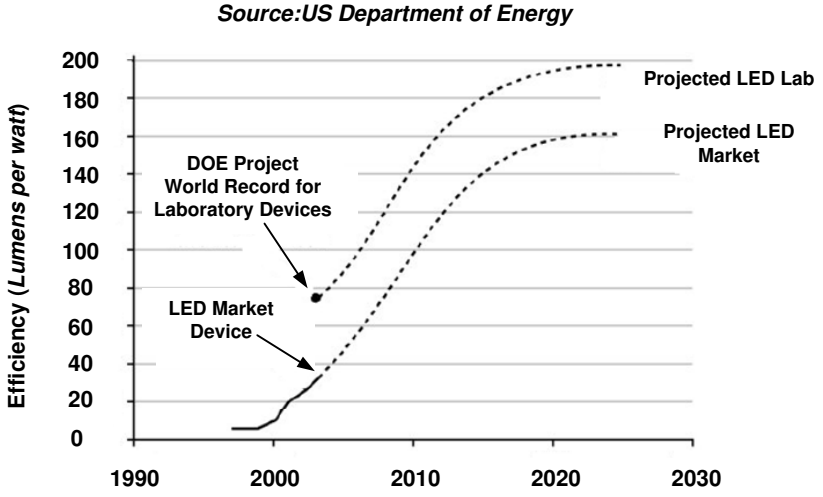

Figure 1.2: Future trends in LED device efficiency

over those using filament lamps. The time from current flow to light output in an LED is measured in nanoseconds. In a filament lamp the response time is about 300 ms. This enables other drivers to quickly see critical signals, e.g., brakes [10].

1.5 LEDs Characteristic Behavior

A LED can be described as a constant voltage load. The voltage drop depends on the internal energy barrier required for the photons of light to be emitted and this energy barrier defines color of light [10]. The I-V characteristic of a LED is similar to a normal p-n junction diode. Below the turn-on threshold voltage, very little current will flow through it. Above threshold, current flow increases rapidly for incremental increases in forward voltage. The threshold for typical automotive, white LEDs is approximately 3.2 Volts [1]. The rise in forward current is exponential with respect to applied voltage above the threshold. Thus the LED can be modeled as a voltage source in series with a resistor. This is valid only at a single operating DC current, as the LED is non linear. If the DC operating current in the LED is changed, then the resistance of the model must be changed to reflect the new operating point. Additionally, at large forward currents, the LED operates at a high power level, which in turn begins to heat the die. As a result, the LED's forward voltage drop decreases and its dynamic impedance starts increasing with respect to temperature. It is important to consider the thermal environment when the LED's impedance is determined [16]. Moreover, lighting characteristics of LEDs, e.g., wavelength shift and luminance decay, are related

with its junction temperature. The thermal interaction of each LED will cause change in color [17]. The I-V characteristic curve of an automotive LED module from OSRAM (LE UW D1W1 01) is shown in figure 1.3 [1]. The I-V curve is non linear and current increases exponentially beyond the forward threshold voltage. The slope of this curve at a specific operating DC current is a measure of the dynamic impedance.

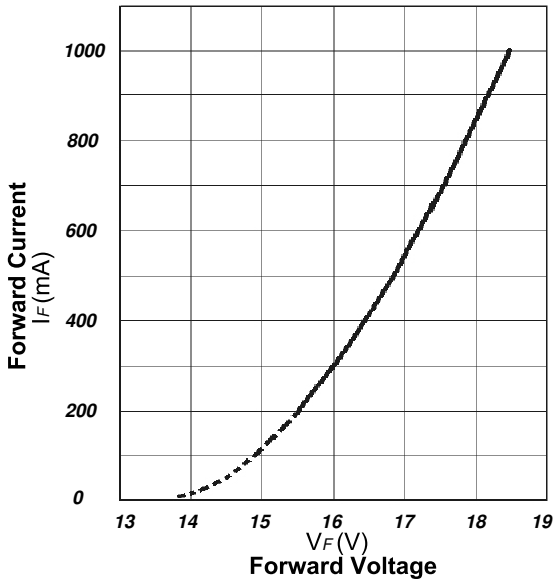

Figure 1.3: Characteristic of an Ultra White Automotive LED [1]

Power LEDs are used in automotive headlights. These headlights have slightly different behavior than normal LEDs; these are current controlled devices compared to one used in interior and back lighting. The output luminous flux is determined by the forward current through them. That is why we need to control current through LEDs [18]. This fact need to be considered while modeling and implementing the control algorithm.

1.6 Comparison Between Linear and Switch Mode Control

Two of the most common methods to drive LEDs are linear and switch mode converters. Linear converters are easier to control. A typical linearly controlled LED is shown in figure 1.4, referenced from [2]. The control circuitry must monitor the output voltage, and adjust the voltage driving gate to hold the output voltage at the desired value. In this case, a field effect transistor (FET) is operating in its

linear conduction mode, as compared to saturation mode in switch mode power supplies (SMPS). Moreover, the FET is conducting continuously through operation, while in SMPS the FET only conducts during its on-time, which in turn depends on the duty cycle. An increased conduction time of the FET in a linear regulator results in more losses. Consequently, low efficiency is one of the major drawbacks. Moreover, in a linear converter, the load voltage should be always lower than the supply voltage. While the SMPS stage can drive a load with a voltage greater than supply voltage, driving LEDs from a SMPS stage can give us better efficiency with less power dissipated in the FET. But control of SMPS driver is more complex than linear driver [18].

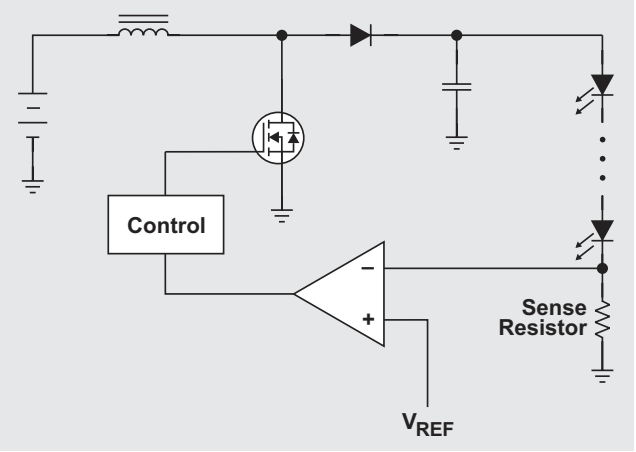

Figure 1.4: Linearly controlled LED setup [2]

Considering the control of both techniques the linear control is very simple. The DC bias to the FET gate is controlled based on a feedback, to maintain constant current. In contrast, control for a SMPS stage is complex. The transfer function of SMPS converters contain a number of poles and zeros. A feedback controlled filter called compensator is designed to cancel out system poles to achieve the desired system gain in the required bandwidth region. The simplest of these compensator designs are far more complex than a linear mode controller. The choice between linear and switch mode power control is a trade off which is driven by efficiency. That is why linear control is suitable for back and interior lighting application, but not for headlights.

1.7 Switch Mode DC/DC Power Converter

An investigation of the most common DC/DC power supply topologies is presented in this section, with the perspective of digital power control. Topologies used in our project are buck, boost and SEPIC; see details below.

1.7.1 Buck Converter

The buck converter is one of the most basic DC-DC switch mode converter. It is a step down converter so its output voltage V_o ranges from the input voltage V_i to 0. The output is not isolated from the input. The input current for a buck power stage is discontinuous or pulsating, due to the power switch (Q1) current that pulses from zero to I_O every switching cycle. The output current for a buck power stage is continuous or non-pulsating, because the output current is supplied by the output inductor-capacitor combination; the output capacitor never supplies the entire load current [19]. A generic implementation of a buck converter is shown in figure 1.5. Current and voltage waveforms are also presented in same figure. The voltage conversion relationship in terms of duty cycle (D) is presented in equation 1.1. The relationship between the output current and the inductor current is presented in equation 1.2.

$$V_o = V_i \times D \tag{1.1}$$

$$I_{L(avg)} = I_O \tag{1.2}$$

1.7.2 Boost Converter

Boost is a non-isolated power stage topology, also called as step-up power stage. Power supply designers choose the boost power stage, because the required output voltage is always higher than the input voltage and has the same polarity. The input current for a boost power stage is continuous, because the input current is the same as the inductor current. The output current for a boost power stage is discontinuous, because the output diode conducts only during a portion of the switching cycle. The output capacitor supplies the entire load current for the rest of the switching cycle [4]. A generic implementation of a boost converter is shown in figure 1.6, along with current and voltage waveforms.

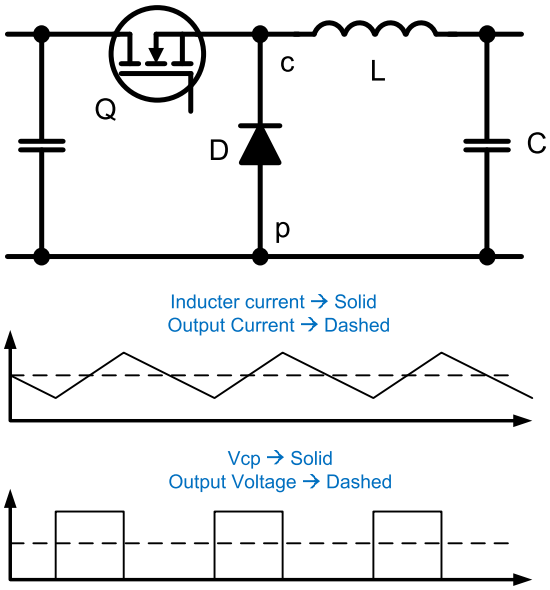

Figure 1.5: Generic buck architecture.

Equation 1.3 shows the voltage translation between input and output voltage. The relationship between output and inductor current in terms of duty cycle (D) is shown in equation 1.4.

$$V_o = \frac{V_i}{1 - D} \tag{1.3}$$

$$I_{L(avg)} = \frac{I_o}{1 - D} \tag{1.4}$$

1.7.3 SEPIC Converter

SEPIC stands for single-ended primary-inductor converter. It can produce an output voltage V_o greater, less and equal to the input voltage V_i. A generic implementation of a SEPIC converter is shown in figure 1.7. The SEPIC converter is similar to a buck-boost converter as both can step-up and step-down voltage. SEPIC has advantages such as isolation and non-inverting output, but has a far complex transfer function that needs higher-order compensator than buck, boost and buck-boost. A voltage translation equation in terms of duty cycle (D) is given

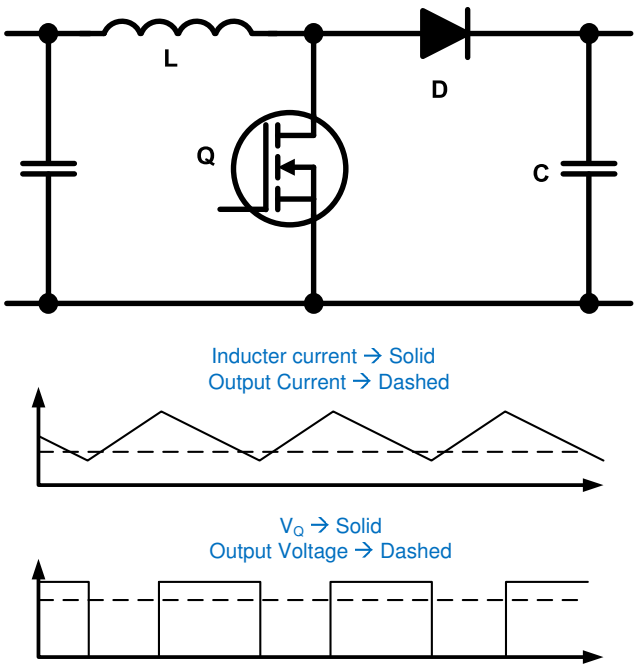

Figure 1.6: Generic boost architecture.

in 1.5.

$$V_o = \frac{V_i \times D}{1 - D} \tag{1.5}$$

1.7.4 Comparison Between Different SMPS Topologies

All converter topologies discussed above are used for different applications depending upon input and output voltage ranges. Apart from this there are some striking differences. Buck and boost converters are non-isolated, while the output of SEPIC is isolated. Isolation can prevent short circuit current from input supply, in case of short circuit at output load. SEPIC is far more difficult to control than buck and boost as its frequency response contains more poles-zeros compared to buck and boost. In addition to the inevitable fourth-order pole of SEPIC, another important feature in the transfer function is a single right half plane (RHP) zero. Right half plane zeros are a result of converters, where the response to an increased duty cycle is to initially decrease the output voltage [20]. Isolation, voltage translation and compensator complexity are major trade-offs faced by the SMPS designer.

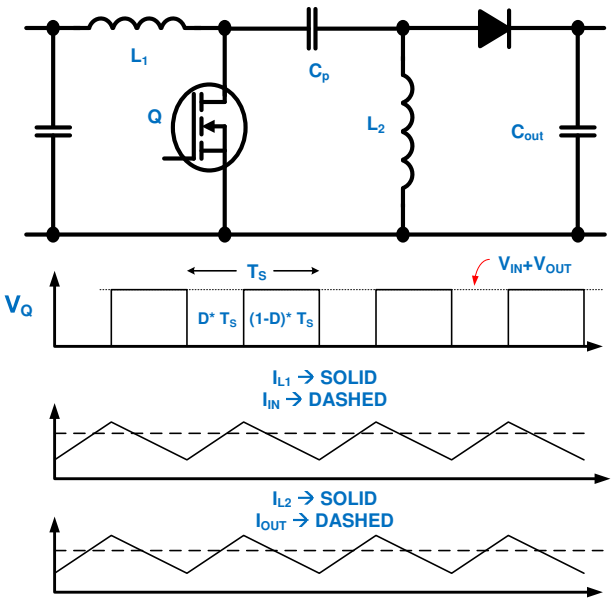

Figure 1.7: Generic SEPIC architecture.

1.8 Comparison Between Analog and Digital Power Management

Power management is an interdisciplinary area of modern electronics, merging hard-core analog circuit design with expertise from mechanical and RF engineering, safety and EMI, materials, semiconductors and magnetic components. Traditionally power supply design is considered an analog trade. But from the very early days, by the introduction of relays and rectifiers, power management is slowly incorporating more and more ideas from the digital world. The introduction of switched mode power conversion required even more digital circuits. Integrated pulse width modulators have introduced even more digital content to power management. Today's highly integrated power management ICs are packed with digital modules, e.g., pulse width modulators and timers. The digital circuits allow the integration of some highly sophisticated features like EEPROM based trimming after packaging to eliminate package stress related initial offsets, digital delay techniques to adjust proper timing of gate drive signals, micro-controllers and state machines for battery charging and management [7].

1.8.1 Analog Controller

Analog control design is traditionally used for switch mode DC-DC converters. A typical block diagram of an analog controller is shown in figure 1.8. It is important to mention that analog compensation blocks in analog controller consist of hardwired circuits and components. This implementation is inflexible and not configurable. That is a major drawback of analog compensation. However, high bandwidth and low response time to input changes are some of benefits in using analog controllers over alternative designs options.

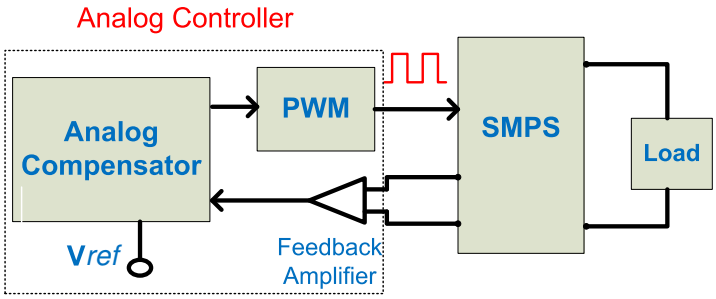

Figure 1.8: Block diagram of SMPS controlled by analog controller

1.8.2 Digital Power Management

Digital power management is a new direction in power supply controller design, to replace the analog circuits by digital implementations. Digital power stands for digital control of the power supply. Digital power supply control attempts to move the barrier between the analog and digital sections of the power supply right to the pins of the control IC [7]. Increased complexity and performance requirements in the discrete analog design of power electronics control have led designers to increasingly consider digital control solutions.

First let us review a digital control model as depicted in figure 1.9. Two major differences from an analog controller are the analog-digital converter (ADC) and the compensator. The analog compensator is here replaced by a digital one, which is either a software implementation executing on a MCU or a hardware solution implemented on an FPGA. Flexibility is a noteworthy advantage of this digital implementation, offering, first, adjustability. That is, every parameter—including voltage and current thresholds, operating frequency, thermal shut down, and startup time—which is measured or programmed can also be adjusted by the digital controller on the fly. Second, flexibility can be used to invoke different control algorithms as the operating conditions of the power supply are changing.

Lastly, due to highly integrated communication peripherals in modern MCUs, the flexibility of the digital approach is greatly enhanced through the on-the fly programmability [7].

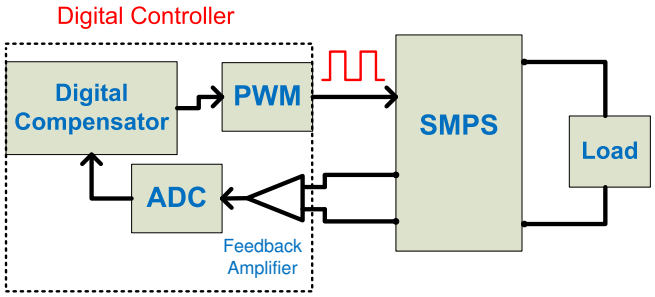

Figure 1.9: Block diagram of SMPS controlled by digital controller

It's important to note that deployment of digital control had no effect on the operating principle and the design of the power stage. The specification of the power supply still determines the choice of topology, the selection of power components and the required control functions. That leaves a fair amount of design tasks still in the analog realm for the power supply experts [7].

1.8.3 Continuous vs Discrete Time Compensation

Analog compensation employs continuous-time S-transforms, while digital compensation employs discrete z-transforms. On the theoretical front, the majority of present implementations translate S-domain transfer functions to Z-domain. This approach permits utilization of the well understood linearized small signal models of switch mode converters. Once the poles and zeros are calculated to ensure the stability of the system, the Z coefficients of the digital transfer function can be found easily. The weakness of this method is that by starting from a linearized model, the benefits of a higher-performance non-linear control theory can not be fully utilized. As a result, the performance of power supplies using either digital or analog controllers are very similar today [7].

1.8.4 Comparison between Analog and Digital Power Management

The striking difference between analog and digital control is the quality and the amount of information available for the controller to make decisions regarding the operation of the power stage. The schemes have their respective advantages and disadvantages. Analog controllers have very high bandwidth and low response

times, while digital controllers are weaker on these parameters in exchange for flexibility and configurability; features that are absent from analog controllers. Table 1.8.4 is formulated to present a comparison between the techniques [7].

Control Properties	Analog	Digital
Switching frequency (CPU limitations)	+	-
Precision(tolerances,aging,temperature effects,drift,offset, etc.)	-	+
Resolution (numerical problems, quantization, rounding, etc.)	+	-
Bandwidth (sampling loop, ADC . DAC speed)	+	-
Instantaneous over current protection	+	-
Compatibility with power components	+	-
Power requirements	+	-
Communication, data management	-	+
Understanding theory	+	-
Advanced control algorithm (non-linear control, improved transient)	-	+
Multiple loops	-	+
Cost of controller	+	-
Cost of a platform (flexibility, time to market)	-	+
Component count(comparable functionality, integration)	-	+
Reliability	+	?

Table 1.2: Comparison of analog and digital controller performance [7]

Chapter 2

Hardware Platform

2.1 Evaluation Module Overview

A LED demo evaluation module (EVM) was available to demonstrate a working demo of LED-based automotive headlights based on the concept of digital power management using a micro-controller. We used the C2000 family MCU from Texas Instruments. The C2000 series is based on a 32-bit CPU, called C2000 or C28x, with different variations of analog and digital peripherals [21]. An important feature is that the EVM is based on the dual inline socket (DIMM), so any of the C2000 series control sticks can be plugged in. We preferred the latest micro-controller available in the Piccolo B family, i.e., TMS320F28035. Development of this EVM was not part of this project but it's worthwhile to mention all the necessary details for the readers to comprehend the firmware development in later part of this report. Broadly an EVM can be divided into two main sections

- SMPS stages
- Central controller

There are four different SMPS stages on our EVM because of two reasons: First, this EVM has been built to support different ranges of LEDs. Different ranges of LEDs require different voltage ranges which in turn are provided by different SMPS converters. Secondly, the purpose of this project is to evaluate and compare between different SMPS topologies with the perspective of digital power control. Certainly in a standard product one has to choose between these converters as some of them as easy to use along with their pros and cons. Figure 2.1 shows a high-level layout of the EVM.

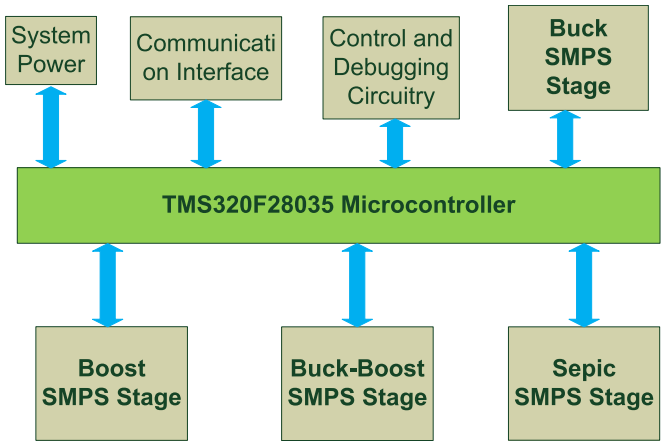

Figure 2.1: General EVM board layout

2.2 SMPS stages

The detailed implementation of different converters is described in subsequent sections. One important fact that applies to all of converters is that there are current feedbacks available from two different points on this EVM; first from the output current and second from the inductor current. These two feedbacks give us the option to design a compensator based on any of these feedbacks.

2.2.1 Buck SMPS Stage

The detailed implementation of a buck SMPS stage is shown in figure 2.2. This is a standard buck implementation as explained in section 1.7.1. Please note that the internal inductor resistance is used as a current shunt for the inductor current feedback.

2.2.2 Boost SMPS Stage

The detailed implementation of a boost SMPS stage is shown in figure 2.3. This is a standard boost implementation as explained earlier in section 1.7.2.

2.2.3 Buck-Boost SMPS Stage

The detailed implementation of buck-boost SMPS stage implementation is shown in figure 2.4. It is important to mention that this is not a standard buck-boost implementation, but actually it is a boost converter whose output load is connected

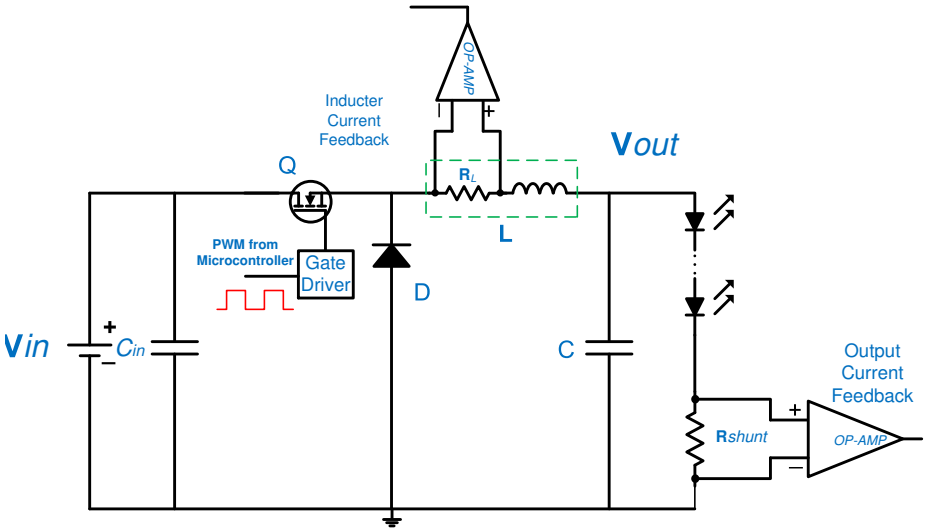

Figure 2.2: Detailed implementation of buck SMPS stage

between the converter's output and input node. Apart from the output load connection, this circuit is the same as the boost converter described in section 2.2.2.

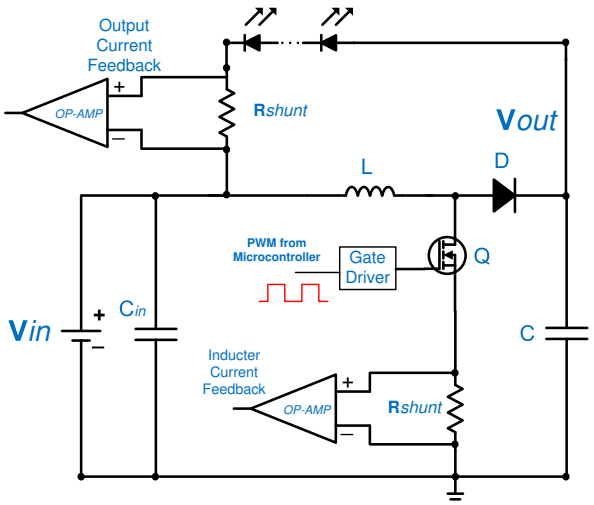

Figure 2.4: Detailed implementation of buck-boost SMPS stage

2.2.4 SEPIC SMPS Stage

The detailed implementation of a SEPIC SMPS stage implementation is shown in figure 2.5. This is a standard SEPIC implementation as explained earlier in section 1.7.3.

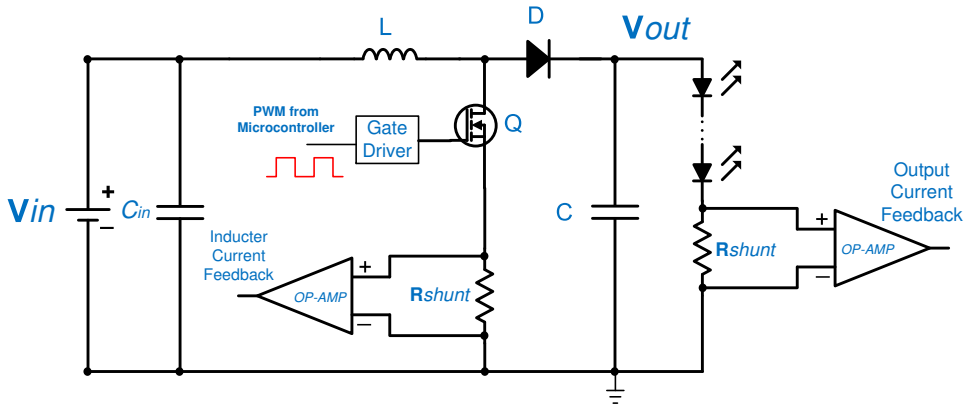

Figure 2.3: Detailed implementation of boost SMPS stage

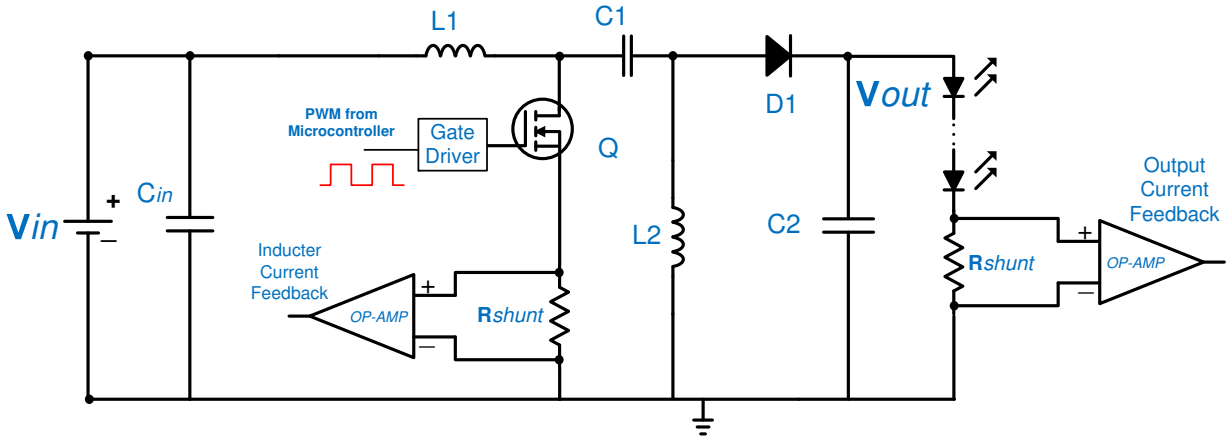

Figure 2.5: Detailed implementation of SEPIC SMPS stage

2.3 Central Controller

The TMS320F28035 micro-controller from the C2000 family is serving as central control unit on the EVM.

2.3.1 TMS320F28035 Real-Time Micro-Controller

The TMS320F2803x family of micro-controllers combines the power of the C2000 processor core and the control law accelerator (CLA) along with highly integrated peripherals. C2000 is a 32-bit processor developed by Texas Instruments. CLA is a 32-bit floating point coprocessor designed to implement control-centric algorithms with details available in section 2.3.2. This MCU is code-compatible with

previous C2000-based controllers. An internal voltage regulator allows for single rail operation. Analog comparators with internal 10-bit references have been added and can be used directly to control the PWM outputs. It also contains a powerful ADC that supports 0 to 3.3 V fixed full scale range conversion and also a ratio-metric conversion based on V_{REFHI}/V_{REFLO} references. The ADC interface has been optimized for low access overhead. A high resolution pulse width modulation (HRPWM) module is available to enable a more precise control[3]. To give a perspective, the block diagram is presented in figure 2.6.

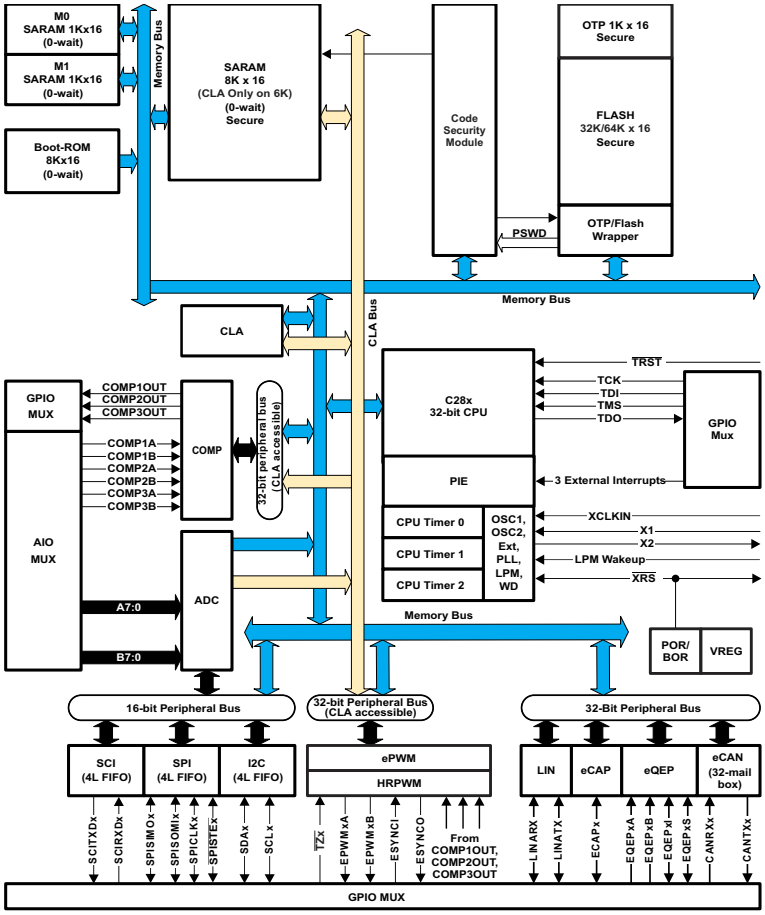

Figure 2.6: Functional block diagram of TM320F28035 [3]

To give a further insight of the integrated peripherals in TMS320F28035, a table 2.3.1 of peripherals is given.

	TMS320F28035
CPU	1 C28x
Peak MMACS	60
Frequency (MHz)	60
CLA	1
RAM (kB)	20
OTP ROM (kB)	2
Flash (kB)	128
PWM	14
CAP/QEP	1/1
ADC	1 12bit
ADC Channels	14
CLA	1
ADC Conversion Time (nsec)	217
I2C	1
UART	1 SCI
SPI	2
CAN	1
Timers	3 32bit GP,
Watchdog Timers	1
GPIO	45
Core Supply (V)	1.8
I/O Supply (V)	3.3

Table 2.1: Detailed list of peripherals in TMS320F28035 MCU [8]

2.3.2 Control Law Accelerator

The CLA is an independent and fully-programmable 32-bit floating-point math processor. As name implies, it brings concurrent control-loop execution capability. It's low interrupt latency allows it to read ADC samples just-in-time. This reduces the ADC sample to output delay to enable faster system response and faster control loops. By using the CLA to service time-critical control loops, the main CPU is free to perform other system tasks such as communications and diagnostics [22].

Functional Overview

The control law accelerator extends the capabilities of the C2000 CPU by adding parallel processing. The CLA is clocked at the same rate as the main CPU. It has

an independent bus architecture consisting of separate data and program buses. It contains an independent eight-stage pipeline. The register file contains four 32-bit result registers (MR0-MR3), two 16-bit auxiliary registers (MAR0, MAR1) and a special function status register (MSTF). Communication between main CPU and CLA is done through two dedicated message RAMs for simplex communication between the CLA and the main CPU. The CLA has direct access to the ePWM 2.3.4, the HRPWM 2.3.4, the comparator and the ADC result registers. This fact makes it very suitable for implementation of control algorithms, also called control laws. Because the control algorithm accesses ADC and PWM registers in every loop iteration, providing access of these registers directly to CLA without any help from main CPU has significantly decreased the access latency. The overall effect would be a faster control loop. The main CPU can map CLA program and data memory to the main CPU space or CLA space.

CLA Instruction Set

The CLA instruction set supports IEEE single-precision (32-bit) floating point math operations. Moreover, some parallel instructions are provided for code optimization. A list below shows all the instructions.

- Floating-point math with parallel load or store
- Floating-point multiply with parallel add or subtract
- 1/X and 1/sqrt(X) estimations
- Data type conversions
- Conditional branch and call
- Data load/store operations

CLA Programming

As the CLA can function independent of the main CPU it executes its own program. The program code for the CLA is written in assembly as there is no compiler available yet. This code is compiled into a separate section using compiler directive. This section is placed in a specific memory bank reserved for the CLA program. The CLA program code can consist of up to eight tasks or interrupt service routines. The start address of each task is specified by the MVECT registers. There is no limit on task size as long as the tasks fit within the CLA program memory space. One task is serviced at a time through to completion. There is no nesting of tasks. Upon task completion a task-specific interrupt is flagged within

the PIE. When a task finishes the next highest-priority pending task is automatically started. Each task can be triggered by following method.

- From C28x CPU via the IACK instruction
- Task1 to Task7 can be triggered by the corresponding ADC or ePWM module interrupts
- Task8 can be triggered by ADCINT8 or by CPU Timer 0

2.3.3 Analog to Digital Converter

The 12-bit ADC module in TMS32028035 is partly based on successive approximation registers (SAR) and partly pipelined. SAR uses a binary search through all possible quantization levels before finally converging upon a digital output for each conversion. A pipeline ADC uses two or more steps of sub-ranging. First, a coarse conversion is done. In a second step, the difference to the input signal is determined with a digital to analog converter (DAC). This difference is then converted finer, and the results are combined in a last step. The exact internal architecture of the ADC is not available due to commercial secrecy.

The ADC runs at full system clock and no pre-scaling is required. The core of the ADC contains a single 12-bit converter which is fed by two sample and hold circuits, which in turn are fed by a total of up to 16 analog input channels. The sample and hold circuits can be sampled simultaneously or sequentially. The converter can be configured to run with an internal band gap reference (0 - 3.3 V) to create true voltage based conversions. It can also use external voltage references V_{REFHI}/V_{REFLO} to create ratio-metric based conversions. This ADC is not sequencer based, however, it is easy for the user to create a series of conversions from a single trigger using software. The basic principle of operation is centered around the configurations of individual conversions called SOC (start of conversion). These SOCs can be configured for trigger, sample window and channel. The ADC can be triggered by multiple sources as listed below [23].

- S/W - software immediate start
- ePWM 1-7
- GPIO XINT2
- CPU Timers 0/1/2
- ADCINT1/2

2.3.4 Enhanced Pulse Width Modulation Module

The enhanced pulse width modulator (ePWM) peripheral is a key element in controlling many of the power electronic systems found in both commercial and industrial equipments. These systems include digital motor control, switch mode power supply control, un-interruptible power supplies (UPS). The ePWM peripheral performs a DAC function, where the duty cycle is equivalent to a DAC analog value. An effective PWM peripheral must be able to generate complex pulse width waveforms with minimal CPU overhead or intervention. It needs to be highly programmable and very flexible while being easy to understand and use. The ePWM unit described here addresses these requirements by allocating all needed timing and control resources on a per ePWM channel basis. Cross coupling or sharing of resources has been avoided, instead each ePWM is built up from smaller single channel modules with separate resources that can operate together as required to form a system. This modular approach results in an orthogonal architecture and provides a more transparent view of the peripheral structure, helping users to understand its operation quickly [24].

Functional Overview

The ePWM module represents one complete PWM channel composed of two PWM outputs, EPWMxA and EPWMxB. Seven ePWM modules are present in this device. Each ePWM instance is identical with one exception. Some instances include a hardware extension that allows more precise control of the PWM outputs. This extension is the high-resolution pulse width modulator (HRPWM). The ePWM modules are chained together via a clock synchronization scheme that allows them to operate as a single system when required. Additionally, this synchronization scheme can be extended to the capture peripheral modules (eCAP). Modules can also operate stand-alone. Each ePWM module supports the following features [24]:

- Dedicated 16-bit time-base counter with period and frequency control
- Two PWM outputs (EPWMxA and EPWMxB) that can be used in the following configurations:
  - Two independent PWM outputs with single-edge operation
  - Two independent PWM outputs with dual-edge symmetric operation
  - One independent PWM output with dual-edge asymmetric operation
- Asynchronous override control of PWM signals through software

- Programmable phase-control support for lag or lead operation relative to other ePWM modules
- Hardware-locked (synchronized) phase relationship on a cycle-by-cycle basis
- Dead-band generation with independent rising and falling edge delay control
- Programmable trip zone allocation of both cycle-by-cycle trip and one-shot trip on fault conditions
- A trip condition can force either high, low, or high-impedance state logic levels at PWM outputs
- All events can trigger both CPU interrupts and ADC start of conversion (SOC)
- Programmable event pre-scaling minimizes CPU overhead on interrupts
- PWM chopping by high-frequency carrier signal, useful for pulse transformer gate drives

High Resolution PWM Module

The HRPWM module extends the time resolution capabilities of the conventionally derived PWM. HRPWM is typically used when PWM resolution falls below 9-10 bits. This module supports both duty cycle and phase-shift control methods. The high resolution capability is only implemented on the A signal path of PWM, that is, on the EPWMxA output. The EPWMxB output has the conventional PWM capability [25].

2.4 Pin Assignments

Signal Name	Description	Target Connection
EPWM-1A	Boost 1	GPIO-00
EPWM-2A	Boost 2	GPIO-02
EPWM-3A	Dimming,Boost1	GPIO-04
EPWM-3B	Dimming,Buck	GPIO-05
EPWM-4A	Buck	GPIO-06
EPWM-5A	SEPIC	GPIO-08
EPWM-6A	Dimming,SEPIC	GPIO-10
EPWM-6B	Dimming,Boost2	GPIO-11
EPWM-7A	-	GPIO-12
EPWM-7B	-	GPIO-13
ADC-A0	Spare ADC Channel 1- Temp Sense	ADC-A0
ADC-A1	Spare ADC Channel 1- Temp Sense	ADC-A1
ADC-A2	Boost1 Inductor Current	ADC-A2
ADC-A3	Buck Output Current	ADC-A3
ADC-A4	Boost2 Inductor Current	ADC-A4
ADC-A5	Buck Output Voltage	ADC-A5
ADC-A6	SEPIC Inductor Current	ADC-A6
ADC-A7	Input Current Sense	ADC-A7
ADC-B0	Boost1 Feedback	ADC-B0
ADC-B1	Boost 1 Output Voltage	ADC-B1
ADC-B2	Boost 2 Output Current	ADC-B2
ADC-B3	Boost 2 Output Voltage	ADC-B3
ADC-B4	Buck Switching Current	ADC-B4
ADC-B5	SEPIC Output Current	ADC-B5
ADC-B6	Input Current Sense	ADC-B6
ADC-B7	SEPIC Inductor Current	ADC-B7

Table 2.2: Pin connection on EVM

It is important to understand the details of connection between SMPS stages and the MCU. These are presented in table 2.4. Please note that PWM modules 3 and 6 are intended to be used for providing dimming PWM signals. Two outputs from each PWM module, i.e., EPWMxA and EPWMxB, are connected to dimming FETs. This setup is designed to control four FETs using two PWM modules. However, it was not possible to provide dimming signal to two different SMPS converters from one PWM module. One PWM module can only control one dimming FET. Since there are only three spare PWM modules, only three converters will be dimmed and one converter will not have dimming capability. Module 3, 6,

7 will be used for dimming boost-1, SEPIC, boost-2 respectively. The buck converter is not dimmed. Since there is no connection between module 7 and boost-2 SMPS, a jumper is placed to enable this modification.

Chapter 3

Design Considerations for LED Automotive Headlights

LEDs are largely used in automotive applications, e.g., in back lighting, interior lighting and panel lighting. Use of LEDs in headlamps was only limited to a few premium cars until recent breakthroughs in technology. LEDs offer a number of advantages, such as far better energy efficiency, over conventional incandescent bulbs which are traditionally used in automotive headlights. This makes them suitable for battery operated systems. A number of design matrices are identified for a digital power controller for automotive LED headlights. This will simplify the design process.

3.1 Design Matrices

Identification of design trade offs early in the design process could help simplify design effort. Some of these are based on theoretical aspects related to modeling and compensation. Moreover, some are related to the MCU-based implementation and the switching frequency. Each of these design trade offs are presented in subsequent sections.

3.1.1 SMPS Modeling

Efficient system modeling is always required for effective control. Control algorithms are built around these models for precise and effective control. Accurate system models are normally more detailed and consequently controls are more complex. The efficiency of a model is directly proportional to its complexity. In practice, some details are left out due to limitations of implementation technology. This also applies to switch mode converters. The converter model complexity di-

rectly translates into higher filter order in the compensator. In an MCU-based implementation this means more instructions in compensator software, that is, longer computation time, that in turn defines the upper limit of switching frequency. Furthermore, a general model is more complex than one tailored to specific operating conditions.

SMPS are inherently non-linear due to the presence of switching devices like FETs and diodes. Averaging approximations can be used to develop linear models of these components [26],[27]. This modeling approach tends to lose important information on system behavior due to averaging approximation. Controller implementations based on this model are thus less efficient. MCU-based implementations give choice of implementing complex and non-linear models. Considering the digital control implementation, its full capability is not used while using linear models. Instead, non-linear models are required to fully exploit the capabilities of an MCU-based digital controller. Modeling SMPS efficiently for specific design specifications and implementation technology could help reducing complexity of plant and consequently the complexity of compensator.

3.1.2 Switching Frequency

The switching frequency in SMPS is one of the most important parameters. SMPS are inherently non-linear due to PWM switching. Small signal stability is ensured by averaging approximations used to model these SMPS in the linear domain. Theoretically such approximations are only valid for half of the switching frequency as proposed by the Nyquist criterion, but in practice the model is far more limited, to 1/10 to 1/7 of the switching frequency. Consequently the compensation designed to control these SMPS is also valid in similar limits. So a system with higher switching frequency can compensate high frequency perturbations. That means that an increased sampling frequency can increase the stability of the system [28]. Furthermore, a higher frequency in SMPS means less output ripple, because higher frequency components are easier to filter out by using low pass filters in the power stage.

In case of digital power management, practical limits are introduced due to MCU-based implementation: The master clock frequency and the sampling times of the ADC are the major limiting factors. The software complexity of the control loop, determined by the underlying SMPS topology, also imposes a limit. For example, the transfer function of a SEPIC converter is more complex than a buck converter, as it contain a fourth-order pole and a right half plane zero. Consequently, a software implementation of the control law for SEPIC will contain more instructions than that for buck, and the control loop for SEPIC will thus take more clock cycles than that for buck. Assuming the same system clock, the repetition frequency of the control loop for SEPIC will be less than that of buck.

Moreover, the switching frequency will determine the bandwidth availability to other software tasks. All these factors make switching frequency an important parameter in design trade offs.

3.1.3 Current Control vs Voltage Control

LED-based automotive headlights behave quite differently than their incandescent counterparts. Traditionally the luminance of incandescent bulbs are controlled by compensating the voltage applied to them. In contrast, LEDs are controlled by compensating the current through them. Voltage-mode control implies that the output voltage to a LED string is controlled and the resulting current, measured by a shunt, is used as a feedback parameter to control the output voltage using PWM duty cycle. This technique is indeed an indirect way to control the current. Current-mode control, also referred as current injection control, uses the fact that the average inductor current has a proportional relationship with average output current. So controlling the average current in an inductor can control the average output current. This is a more direct approach based on current injection into system. Modeling the system in current mode can reduce the system order, consequently reducing the order of the compensator. A lack of documentation available on current mode control in the context of digital power management is a major hurdle in our case. In contrast, voltage-mode control is well known and a lot of published work is available. These two modeling alternative serve as an important trade off; note that the different complexity of the respective compensator has an effect on switching frequency too.

3.1.4 Life Cycle and Diagnostics

Automotive based designs are always constrained by life cycle considerations. By using necessary diagnostic functions, expensive pieces of equipment can be safeguarded from damage. Considering a digital controller, it's easy to implement diagnostics; additional logic is required in the case of analog controllers. In digital power management, control loops and diagnostic functions can share the same MCU using time-sharing techniques. But diagnostics need a fair share from the valuable system bandwidth. System bandwidth is first allocated to the control algorithm to fulfill specifications. The rest of the bandwidth is shared between diagnostics, communication and system monitoring. A comprehensive analysis of relative importance and time-to-action margins for different diagnostic functions is required to estimate bandwidth requirements of different diagnostic functions.

Chapter 4

Modeling and Compensation

Modeling of SMPS converters is the first of the tasks at hand. There are two current feedback alternatives available in each converter circuit 2.2. These two alternative allow us to use two different control algorithms. SMPS topologies implemented on EVM inherently operate in voltage mode. This is called voltage-mode control and implies that the voltage across load is controlled. However, we need to control the current through LEDs. These models need to be modified to control current. Considerable documentation is available on modeling of these converters in voltage mode. This fact make it first choice for implementation. The second solution is to model converters for current-mode control. This is relatively difficult because of lack of existing published work in the context of digital power management. Two approaches are listed below

- In the first technique, the average output current is controlled by using feedback from a output current shunt. The current in the shunt is proportionally related to the output voltage of the converter. This technique is termed voltage-mode control (VCM). Existing voltage-mode models with modifications will be used in this case.

- In the second approach, the output current is controlled indirectly by controlling the input current or the inductor current. This approach makes use of the fact that average output current is proportional to the average inductor (input) current. This technique is termed current-mode control (CCM). A new model will be developed in this case.

4.1 Voltage-Mode Control

In this approach, the output current is controlled by using feedback from the output current shunt in series with the LED. A converter using this feedback operates

in voltage mode i.e. output voltage is varied by changing PWM duty cycle. Significant amount of existing published work and books present such models [29]. These models provide transfer functions of output voltage in terms of control parameter, i.e., the duty cycle. However, we need to maintain current through the LED load. This technique models the converter in voltage mode and just replaces the load with a dynamic LED load model. LEDs have a dynamic resistance behavior so the chosen value of resistance represents a specific output current operating point (Q-point). In the subsequent section a general explanation of this technique is presented and it is applied to all the different SMPS topologies on our EVM.

4.1.1 Modeling of SMPS

Case I

The model derived in this section only applies to buck, boost, SEPIC converter on EVM because this models considers the LED's load connected between output node and ground. Consider the SMPS stage driving a LED load in figure 4.1. There is a current shunt to sense the output current value. The current through the LEDs depends upon the voltage applied, which in turn depend on the duty cycle. The transfer function of the output current in terms of duty cycle will be calculated. Existing transfer functions on voltage to duty cycle are used to find the desired transfer function. Let us consider this duty cycle to output voltage transfer functions as $G_v(s)$.

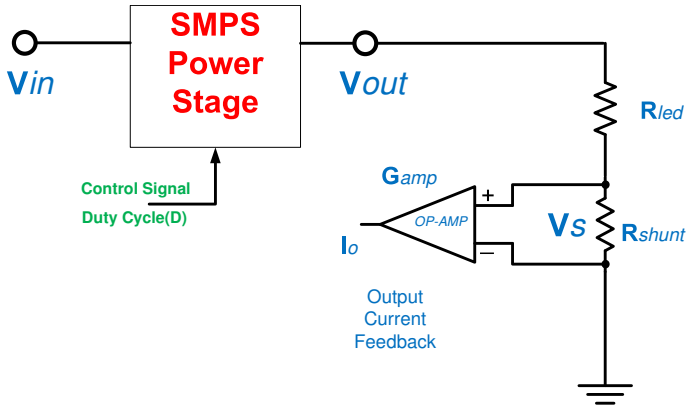

Figure 4.1: Voltage control mode

$$G_v(s) = \frac{V}{D} \tag{4.1}$$

Considering the general output stage of converters in figure 4.1, a transfer function from output current to duty cycle will be computed in the following steps.

The total output resistance seen by the converter is

$$R = n \times R_d + R_s \tag{4.2}$$

where

- Total output resistance = R
- LED forward biased resistance = R_d
- Current shunt resistance = R_s
- Number of LEDs = n

So the current through the LEDs will be

$$I_o = \frac{V_o}{R} \tag{4.3}$$

This current produces a voltage in the current shunt that is given by

$$V = \frac{V_o \times R_s}{R} \tag{4.4}$$

This voltage is a measure of the current through the LEDs and will be amplified by the current sense amplifier. Later it will be sampled by the analog to digital converter for current feedback. A transfer function from output current to output voltage can be written as

$$G_i(s) = \frac{I_o}{V_o} = \frac{G_{amp} \times R_s}{R} \tag{4.5}$$

A current to control transfer function can easily be found now

$$G(s) = G_i(s) \times G_v(s) = \frac{I_o}{V_o} \times \frac{V_o}{D} = \frac{I_o}{D} \tag{4.6}$$

Our model proposes a product of an already existing output voltage to duty cycle transfer function $G_v(s)$ with a newly derived $G_i(s)$ to get an output current

to duty cycle transfer function. Note that $G_i(s)$ does not have any frequency-dependent component and its just a DC scaling value. So the existing voltage-mode model will be scaled with a value to get the current model.

In subsequent sections we will apply this model to all our underlying converters. A Matlab script is used to implement these models. The theory explained in [30] is used to develop this script. Initially a continuous-time model is implemented. Later a corresponding discrete model is developed using a continuous-to-discrete transform function available in Matlab.

Case II

The buck-boost converter 2.2.3 on our EVM is not a conventional buck-boost converter, but it is a boost converter with its output load connected between output and input node. This circuit configuration enabled step-up and step-down operations in boost converters similar to the buck-boost type. The model derived in the last subsection is not valid in this case as the previous model assumes an output load connected between output node and ground. Another model has to be derived:

Considering the LEDs are connected between output and input 2.4, the current through the LEDs will be

$$I_o = \frac{V_o - V_{in}}{R} \tag{4.7}$$

This current produces a voltage in the current shunt

$$V = \frac{(V_o - V_{in}) \times R_s}{R} \tag{4.8}$$

This voltage is a measure of the current through the LEDs and will be amplified by a current sense amplifier. Later it will be sampled by the analog to digital converter for current feedback. A transfer function from output current to output voltage can be written as

$$G_i(s) = \frac{I_o}{V_o} = \frac{1 - \frac{V_{in}}{V_o} \times G_{amp} \times R_s}{R} \tag{4.9}$$

Note that our model is valid for small signal perturbations around a DC operating point (Q-point). The duty cycle value is constant on this Q-point, so the term $\frac{1-V_{in}}{V_o}$ can be replaced with duty cycle (D). The equation 4.9 thus will become

$$G_i(s) = \frac{I_o}{V_o} = \frac{D \times G_{amp} \times R_s}{R} \tag{4.10}$$

Hence equation 4.10 is a counterpart of equation 4.5.

Buck Converter

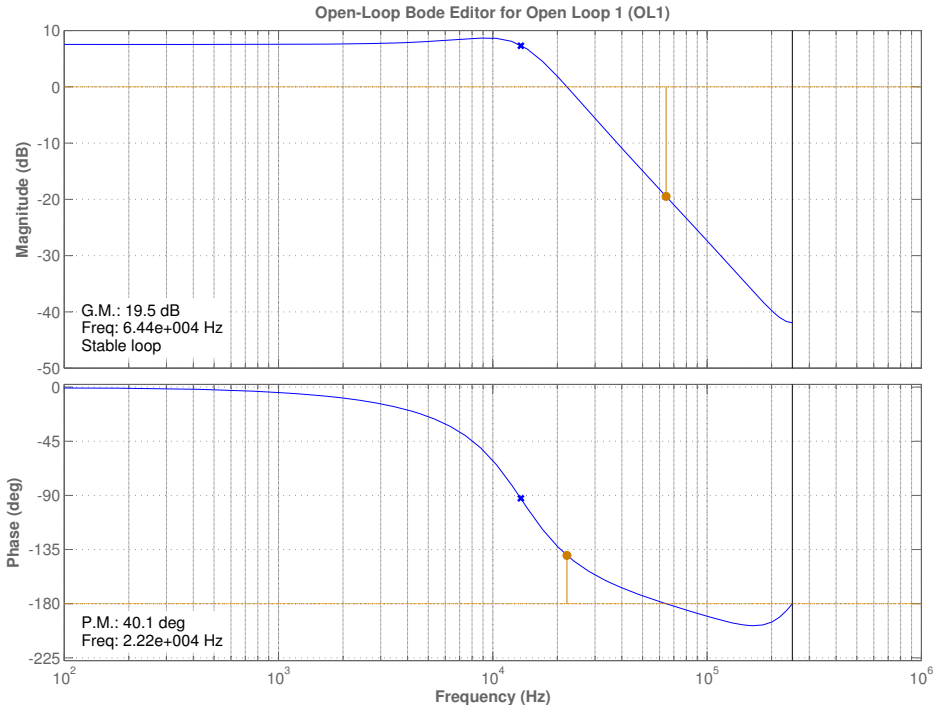

Figure 4.2: Bode plot of Matlab model for buck converter

The voltage-mode transfer function for the buck converter is extracted from [19]. This model is presented in equation 4.11.

$$G_v(s) = V_i \times \frac{R}{R+R_L} \times \frac{1 + R_C \times C}{1 + S \times [C \times (R_C + \frac{R \times R_L}{R+R_L}) + \frac{L}{R+R_L}] + S^2 \times (L \times C \times \frac{R+R_C}{R+R_L})} \tag{4.11}$$

The final transfer function will be calculated by using equation 4.5 and 4.6 from above mentioned model. This model is implemented in a Matlab file in section A.1. The bode plot for this model is shown in figure 4.2.

Boost Converter

The voltage-mode transfer function for the boost converter is extracted from [4]. This transfer function is presented in equation 4.12. The final transfer function will be calculated by using equation 4.5 and 4.6 from above mentioned model.

This model is implemented in a Matlab file in appendix A.2. The bode plot of this model is shown in figure 4.3.

$$G_V(s) = G_{do} \times \frac{(1 + \frac{S}{W_{Z1}}) \times (1 - \frac{S}{W_{Z2}})}{1 + \frac{s}{W_O \times Q} + \frac{S^2}{{W_O}^2}} \tag{4.12}$$

$$G_{do} = \frac{V_I}{(1-D)^2} \tag{4.13}$$

$$W_{Z1} = \frac{1}{R_C \times C} \tag{4.14}$$

$$W_{Z2} = \frac{(1-D)^2 \times (R - R_L)}{L} \tag{4.15}$$

$$W_O = \frac{1}{\sqrt{L \times C}} \times \sqrt{\frac{R_L + (1-D)^2 \times R}{R}} \tag{4.16}$$

$$Q = \frac{W_O}{\frac{R_L}{L} + \frac{1}{C \times (R + R_C)}} \tag{4.17}$$

Buck-Boost Converter

The buck-boost converter on our EVM has a similar circuit to the boost converter in section 4.1.1. The only difference is that the LED load is connected between the output and input node. The voltage-mode transfer function presented in a previous section 4.1.1 will be used. The final transfer function is computed using equations 4.10 and 4.12. The detailed Matlab implementation is shown in appendix A.3. The bode plot of this model is shown in figure 4.4.

SEPIC Converter

The voltage-mode transfer function is extracted from [20]. This is presented in equation 4.18. This model is implemented in a Matlab file A.4. The bode plot for this model is presented in figure 4.5.

$$G_V(s) = \frac{1}{D'^2} \times \frac{(1 - S \times \frac{L_1 \times D^2}{R \times D'^2})(1 - S \times \frac{C_1(L_1+L_2) \times R \times D'^2}{L_1 \times D^2} + S^2 \times \frac{L_2 \times C}{D})}{[1 + \frac{S}{W_{O1}Q_1} + \frac{S^2}{(W_{O1})^2}][1 + \frac{S}{W_{O2}Q_2} + \frac{S^2}{(W_{O2})^2}]}$$

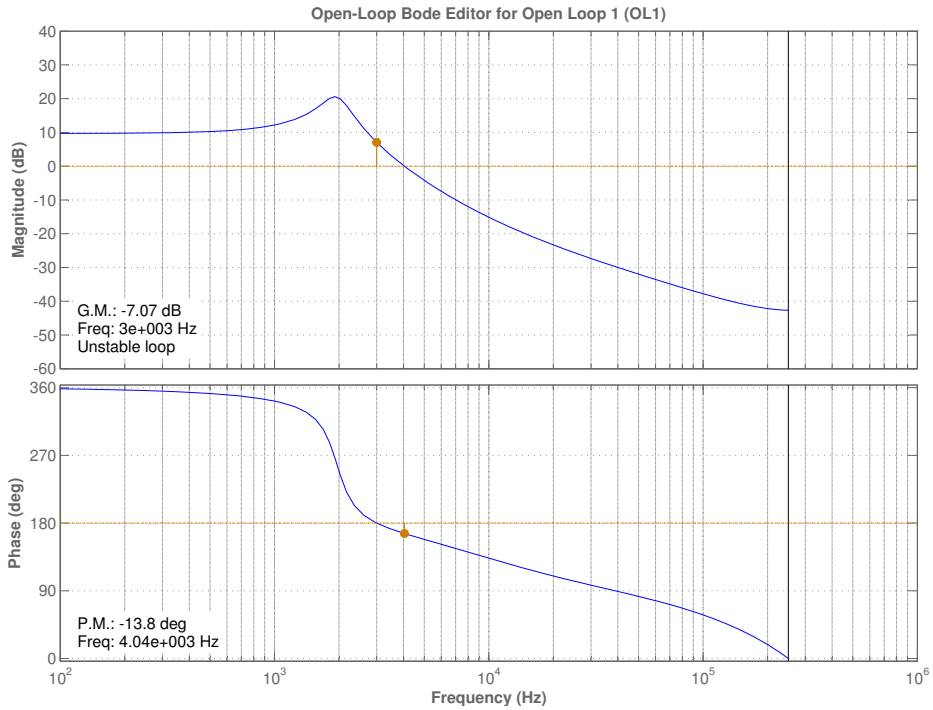

Figure 4.3: Bode plot of Matlab model for boost converter

(4.18)

$$W_{O1} = \frac{1}{\sqrt{L_1[C_2 \times \frac{D^2}{D'^2}] + L_1(C_1 + C_2)}} \tag{4.19}$$

$$Q_1 = \frac{R}{W_{O1} \times [L_1 \times \frac{D^2}{D'^2} + L_2]} \tag{4.20}$$

$$W_{O2} = \sqrt{\frac{1}{A_0} + \frac{1}{A_1}} \tag{4.21}$$

$$A_0 = \frac{L_1 \times C_1 \times C_2}{(L - 1 \times C_1) + C_2} \tag{4.22}$$

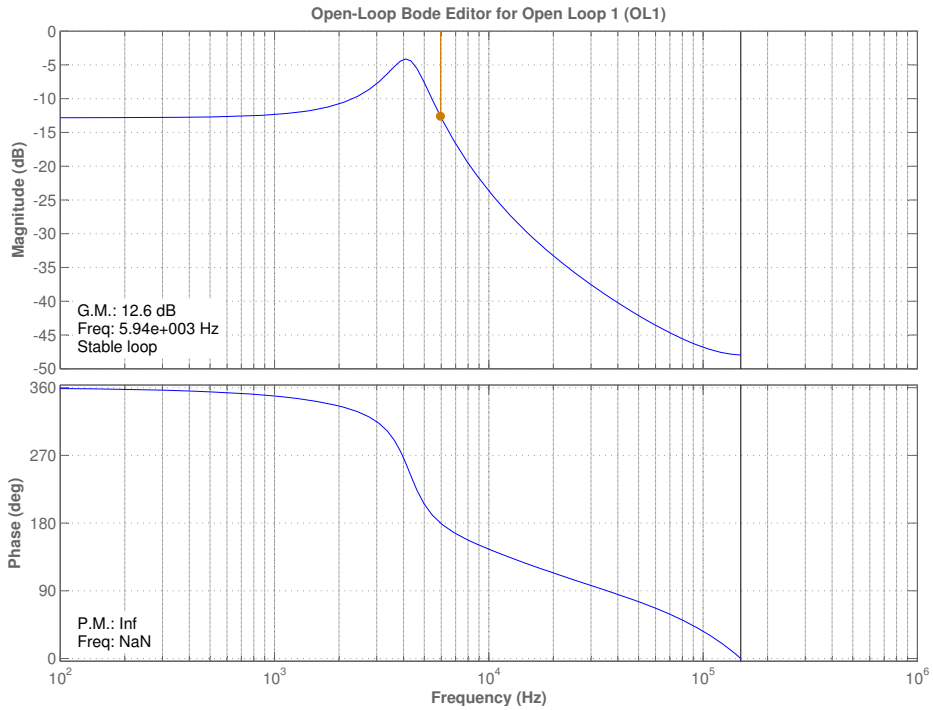

Figure 4.4: Bode plot of Matlab model for buck-boost converter

$$A_1 = \frac{L_2 \times C_1 \times C_2}{(L_2 \times C_1 \times D_2) + (C_2 \times D_1)} \tag{4.23}$$

$$Q_2 = \frac{R}{W_O 2(L_1 + L_2)\frac{C_1(W_{O1})^2}{C_2(W_{O2})^2}} \tag{4.24}$$

4.1.2 Compensation

Compensation in a control system is implemented to ensure stable operation. The compensator does this by minimizing some critical system parameters. Phase lag at crossover frequency is called phase margin. Gain lag at the point where the phase crosses 180 degrees is called gain margin. Lower values of gain and phase margins are desired for closed-loop system stability. Good values for phase margin are in the range of 45 to 60 degrees and 6 to 10 dB for gain margin. The Single Input Single Output (SISO) tool in Matlab is used to design the compensator, and design flow for this can be summarized as follow:

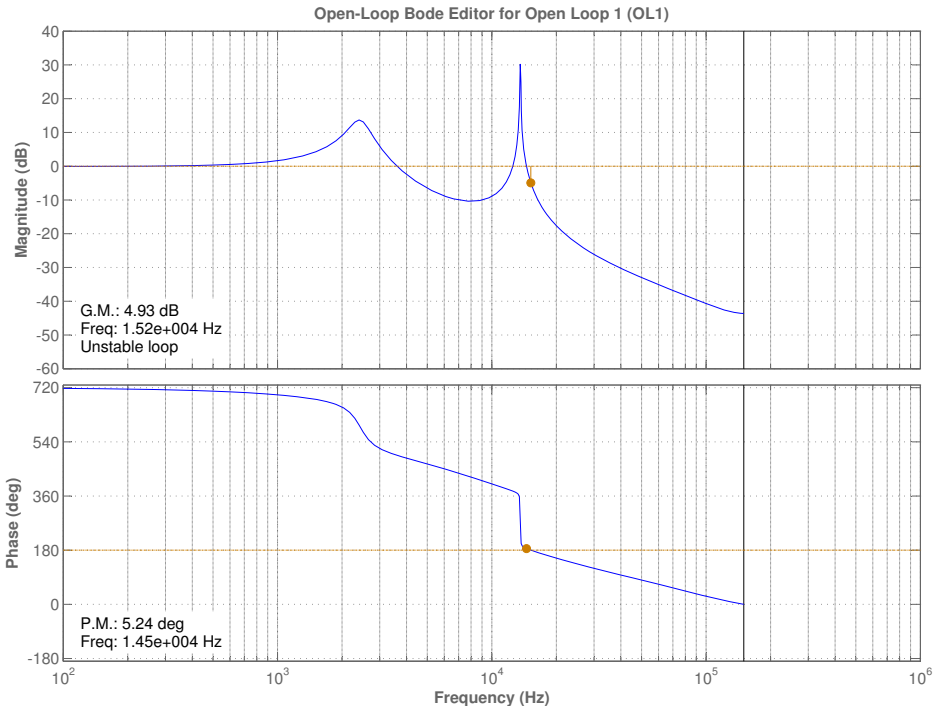

Figure 4.5: Bode plot of Matlab model for SEPIC converter

- Discrete model developed in Matlab is imported into SISO design tool
- A pole is inserted at low frequency for keeping high gain at low frequency and roll off at high frequency
- Insert zeros in proximity of converter poles to compensate
- Insert a pole in high frequency in proximity of Nyquist frequency ($\frac{f_{sample}}{2}$) to reduce gain beyond $\frac{f_{sample}}{2}$
- Moderate values for stability parameters, i.e., gain and phase margin, are achieved by adjusting gain and position of poles and zeros
- Finally the compensator transfer function is exported to Matlab

First, the SISO design tool is used to design a compensator. Later the coefficient values for an infinite impulse response (IIR) filter is calculated. This filter is implemented in a micro-controller. A Matlab script A.5 is written to automate this process. The bode plot of the compensated model for the buck converter using the SISO tool is shown in figure 4.6.

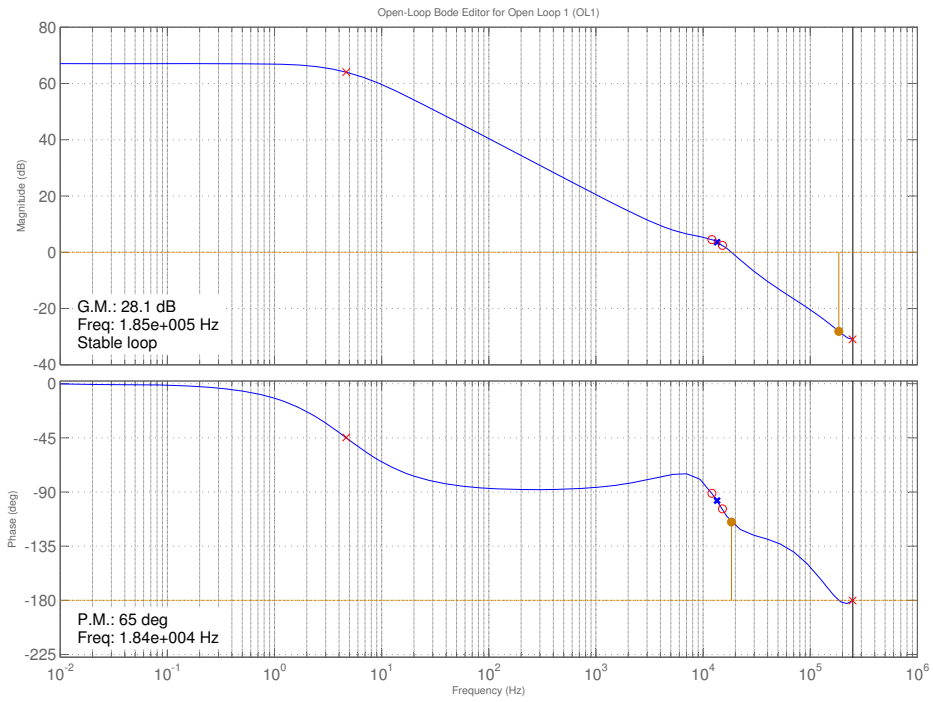

Figure 4.6: Bode plot of compensated Matlab model for buck converter

The bode plot of the compensated model for the boost converter, using the SISO tool, is shown in figure 4.7.

4.2 Current-Mode Control

As a second proposal, the output current can be controlled by controlling the inductor current or the input current. This method is called current injection mode control [31]. Considerable documentation is available for analog controllers, however, information on digital controllers employing this technique is scarce. The relation between the average inductor current and the average output current is shown in equation 1.4. So by controlling the average inductor current we can control the average output current [32]. The inductor current feedback is also available on the EVM. This approach will help to reduce the order of the transfer function. Consequently the compensator will be of lower order consuming less instructions and time. Consider the boost converter presented in section 1.7.2. During the on time of PWM cycle, the switch Q is turned on and the input voltage V_{IN} is applied to inductor L. Consequently there will be an linear increase in

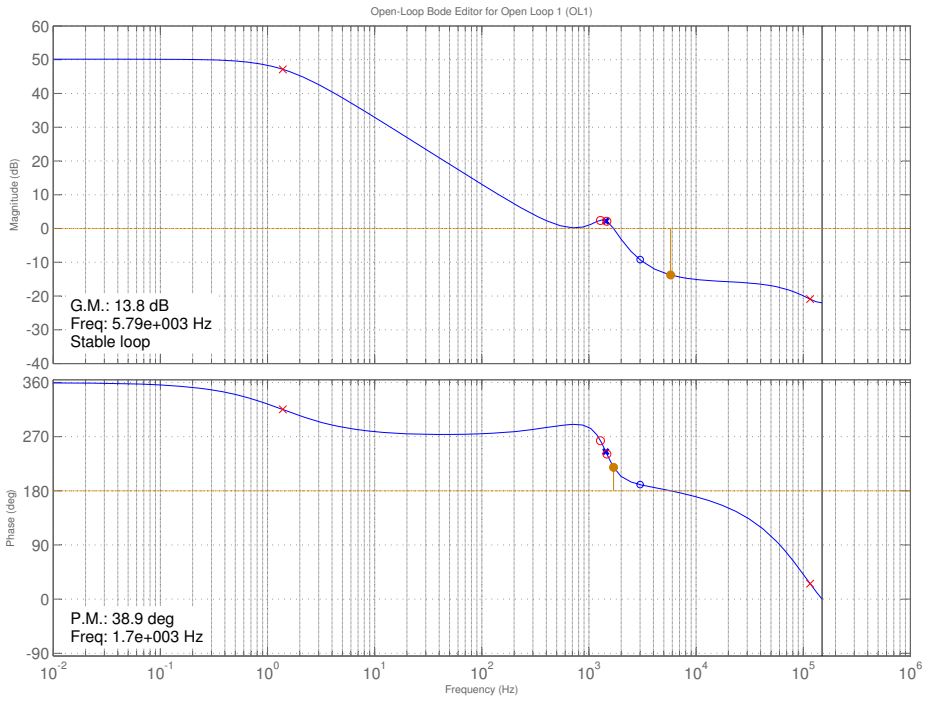

Figure 4.7: Bode plot of compensated Matlab model for boost converter

current. The slope of this current will depend upon input voltage and inductance. During the off time of PWM cycle, when the switch in turned off, the inductor discharges through the output load. Both these situations are depicted in figure 4.8. The voltage and current waveforms are presented in figure 1.7.2.

The average current through the inductor can be maintained by controlling the midpoint of the current rising waveform. That means that if the inductor current is sampled at midpoint of the PWM on cycle, this sample corresponds to the average inductor current. Compensating this value with a required reference is desired. In short, the behavior of the inductor and the rest of the power stage becomes irrelevant in off time of PWM, which leaves us only one frequency dependent element, that is, the inductor. So there will be only one pole in the system with a value of $\frac{L}{R}$. The average current through the inductor during the whole PWM cycle will depend upon the duty cycle. So a change in duty cycle will change the average inductor current. The current in inductor can be given as

$$\triangle I_L = \frac{V_in}{L} \times \triangle T \tag{4.25}$$

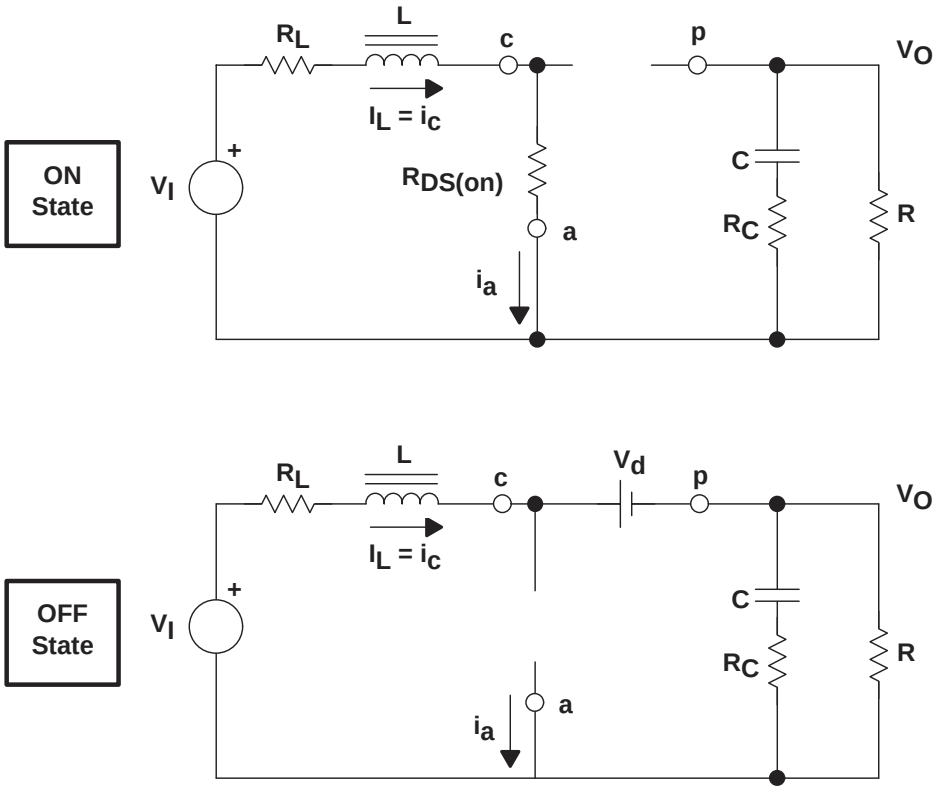

Figure 4.8: Two switching states of boost converter [4]

The current rise in on time of PWM cycle

$$\triangle I_L = \frac{V_i n}{L} \times T_{ON} \tag{4.26}$$

The average inductor current is the mid-point of this current ripple so the average current can be represented as

$$I_{avg} = \frac{\triangle I_L}{2} \tag{4.27}$$

moreover

$$T_{ON} = D \times T_S \tag{4.28}$$

$$I_{avg} = \frac{V_{IN} \times T_s}{2 \times L} \times D \tag{4.29}$$

$$\frac{I_{avg}}{D} = \frac{V_{IN} \times T_s}{2 \times L} \tag{4.30}$$

Equation 4.30 gives the steady state behavior. Replacing L with Z_L will give us the final transfer function, where

$$Z_L = R_L + s \times L \tag{4.31}$$

$$G(s) = \frac{I_{avg}}{D} = \frac{V_{IN} \times T_s}{2 \times Z_L} \tag{4.32}$$

$$G(s) = \frac{V_{IN} \times T_s}{2 \times (R_L + s \times L)} \tag{4.33}$$

$$G(s) = \frac{V_{IN} \times T_s}{2 \times R_L \times (1 + s \times \frac{L}{R_L})} \tag{4.34}$$

This model is implemented in a Matlab script that is presented in section A.6. The bode plot for this is shown in figure 4.9. Please note that this model is not linear. Benefits of non-linear control implementation on MCU can only be reaped by using non-linear models. Further work is suggested to refine this model.

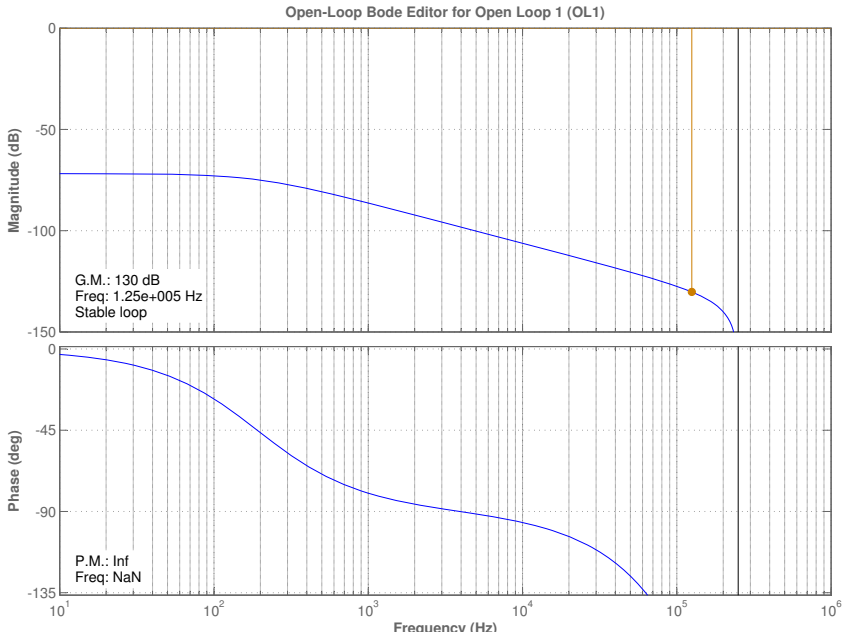

Figure 4.9: Bode plot for current mode control model for boost converter

Chapter 5

System Architecture and Firmware

5.1 Software Development Tools and Resources

The Code Composer Studio (CCS) Integrated Development Environment (IDE) is used for development and debugging of software, which is written in C and assembly languages. The firmware for the C28x main CPU is written in C, while CLA code is written in assembly due to un-availability of compiler. The XDS100 emulator from (Texas Instruments) TI and the XDS510 emulator from Spectrum Digital is used for debugging software. A watch window available in the CCS environment is used as GUI for monitoring system parameters and user input. CCS3.3 was initially used, but CCS4 is used later on in the development cycle. A number of available software libraries and frameworks are used for system development. Some of these were optimized later according to project requirements.

5.1.1 Digital Power Library

As part of a package control suite from TI, the Digital Power Library (DPL) is a set of drivers available to designers to ease application development. Details of this library can be found in [33]. These libraries are optimized in term of instruction count and their use can give a head start in the development phase. We used two types of driver in our implementation: Control filters and PWM update driver.

5.2 System Overview

Challenges related to development and architecture of firmware are presented in this chapter. Careful planning was required to divide the firmware into suitable partitions. Moreover two other issues needed to be addressed: First, because of availability of two CPUs, the allocation of different software parts to two CPUs

requires an early system analysis. Secondly, an allocation of bandwidth of different software functions is also an important factor as total system bandwidth is constant. The obvious choice for implementation of control loop would be CLA 2.3.2, while diagnostics and monitoring functions are implemented on the C28x CPU.

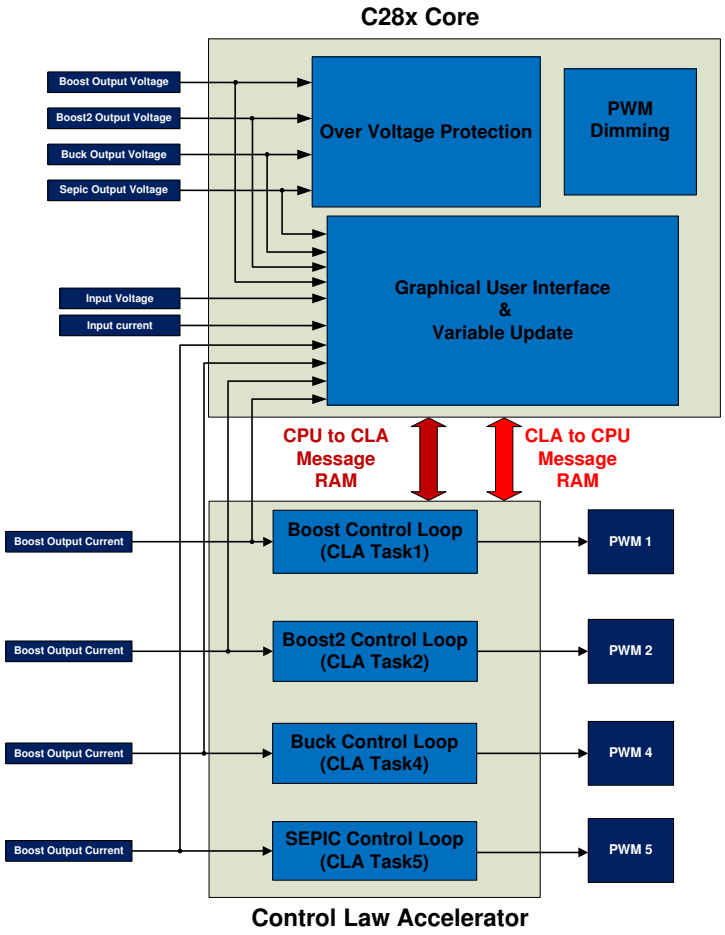

Figure 5.1: General software architecture.

The software implementation can be divided into several main categories:

- SMPS control loop
- Diagnostics function
- Variable monitoring and update

- PWM dimming
- Graphical User Interface
- In car communications

Four of the above mentioned software parts are implemented in this project, while the last one will follow in future versions. A generalized view, showing implementation of different software elements, is shown in figure 5.1. As can be seen, four control loops will be implemented on the CLA. Functions implemented on the C28x CPU include over-voltage protection, dimming and GUI.

Since the evaluation of a number of alternative algorithms is the very purpose of this project, a number of software version are developed. A modest version of code is presented in appendix B. The complete software package is provided at `http://www.cse.chalmers.se/edu/course/Digital_Power_Control_Automotive_LED_Headlights/`.

A detailed explanation will follow in subsequent sections.

5.3 Challenges Related to Discrete System Implementation

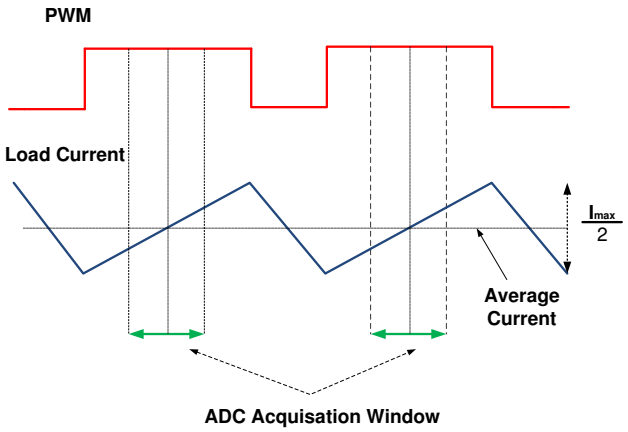

Figure 5.2: Sampling true average current

Considering the discrete-time system implementation, the particular instant at which feedback will be sampled during the whole control cycle have considerable importance. The LED current is supposed to be controlled. However, the output current waveform rises and falls during a PWM cycle. Moreover, we want to maintain the average current through LEDs, which also requires sampling at

specific instants. In short, the sampling instant becomes very important in a digital controller.

In order to control the average current, the feedback will be sampled at the midpoint of the PWM on cycle. Note that the control loop will not have any effect from this implementation. The control loop is just minimizing the error between the feedback and the reference. This phenomenon is depicted in figure 5.2. One important point worthwhile to mention is that the above discussion applies to both voltage- and current-mode control. In current-mode control, it becomes even more important as the inductor current exhibits higher ripple slope than the output current does. A detailed software implementation of this is presented in appendix B.1.

5.4 Hardware Initialization

The initialization of the MCU and different peripherals is explained in this section. There are some general initializations related to MCU that are needed to be done: The master clock period should be defined. GPIOs are configured as input, output or special function, i.e., PWM . The clock signal to different hardware modules is enabled depending upon requirement and so on. There are some initializations that are closely related to digital power management as one defines system behavior listed in the previous section 5.3. Others are described in subsequent sections.

5.4.1 PWM Initialization

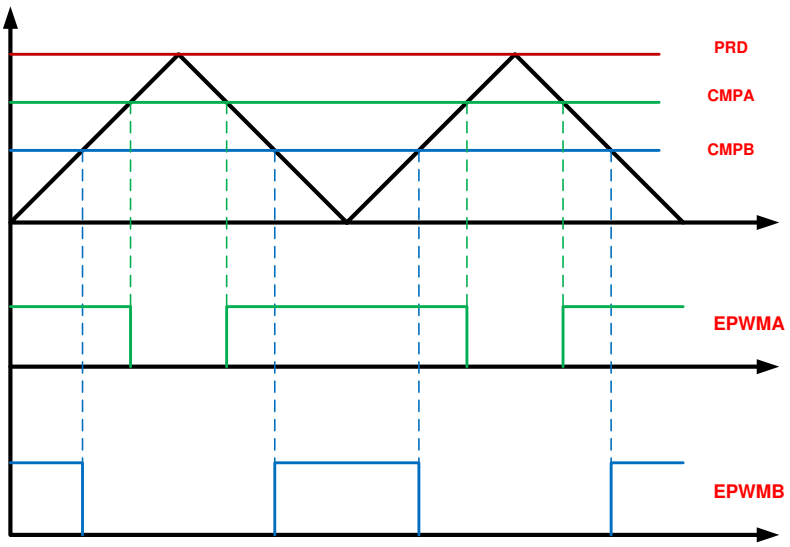

Figure 5.3: Symmetric PWM waveform

The PWM module is configured in symmetric mode. This mode is illustrated in figure 5.3. The time base counter is configured in count up and down mode. Compare A and B register is used to generate PWM signal. PWM module 1 is configured as master and module 2, 4, 5 are configured as slave with a phase difference of 90, 180, 270 degree respectively. This is graphically presented in figure 5.4. Details of this firmware is presented in appendix B.6. This file also contains configuration of dimming PWM module. Initialization of PWM modules 3, 6, 7 for driving dimming FET is the same with some difference. Dimming PWM modules operate in non synchronous mode.

5.4.2 ADC Initialization

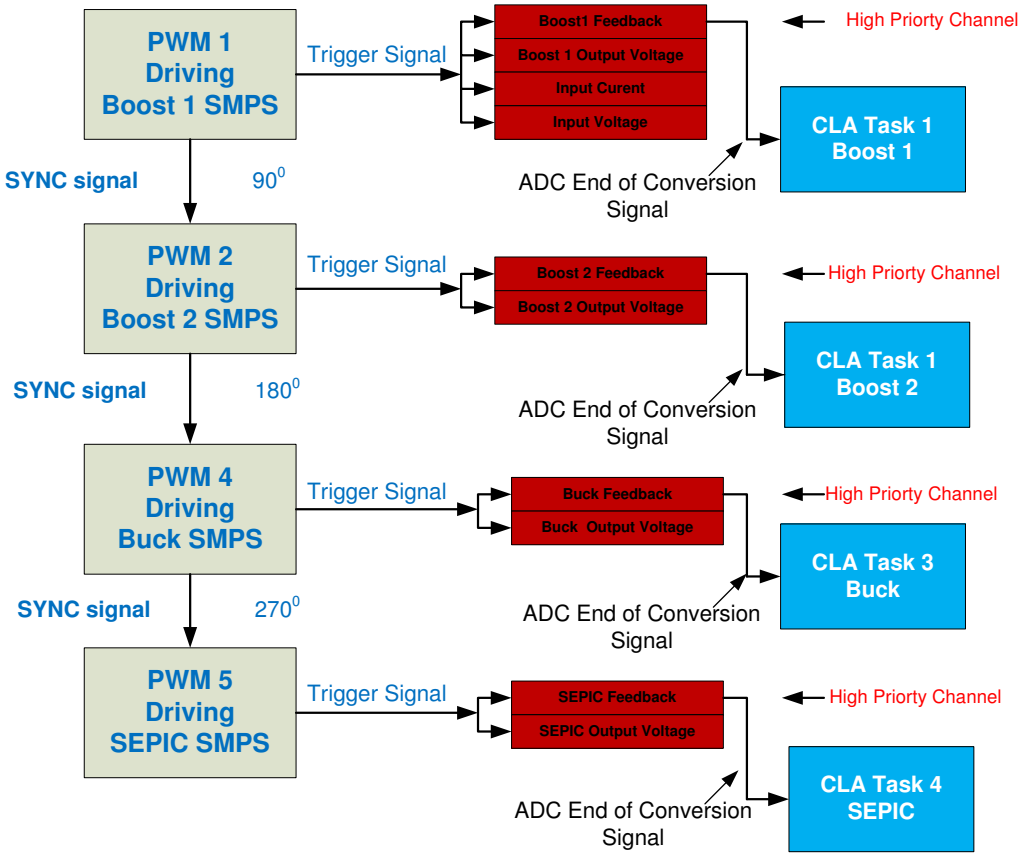

Figure 5.4: Synchronization of PWM and triggering of ADC channels

Important parameters in ADC initialization are triggering, acquisition window and priority of different channels. Moderate acquisition times are tried and fixed to 8 clock cycles as it does not have any considerable effect on performance. Channels connected to output current feedbacks of four converters are given high priority. Each of these channels is triggered by the corresponding PWM modules driving these SMPS converters. For example, PWM 1 is driven by PWM1. ADC channel sampling boost 1 output current is triggered by PWM1. This scenario is depicted in figure 5.4. Other signals associated with same converter will also be triggered by PWM 1, however they are configured as low priority SOCs so they will be computed afterwards. Since PWMs have 90 degree phase shift from previous PWM module, there will be enough time for low priority channel's conversion.

5.4.3 CLA Initialization

The CLA being a coprocessor requires significant initializations after which it can work independently. Initializations required by the CLA module are listed below:

- Program and data memory is needed to be allocated to CLA
- Message RAMs are initialized with variables for inter processor communications
- Required number of tasks are enabled
- Triggering to task is defined

Since the control law is only required to be computed when a new feedback sample is available, the CLA tasks are triggered by the end of the conversion signal from the ADC channel connected to the output current feedback. This is depicted in figure 5.4.

5.5 Control Loop Implementation

The control loop is the most important of all software parts. This is implemented on the CLA as it is optimized for realtime control loop implementations as discussed in section 2.3.2. A typical control law implementation consists of three steps. A control flow graph is showing this in figure 5.5.

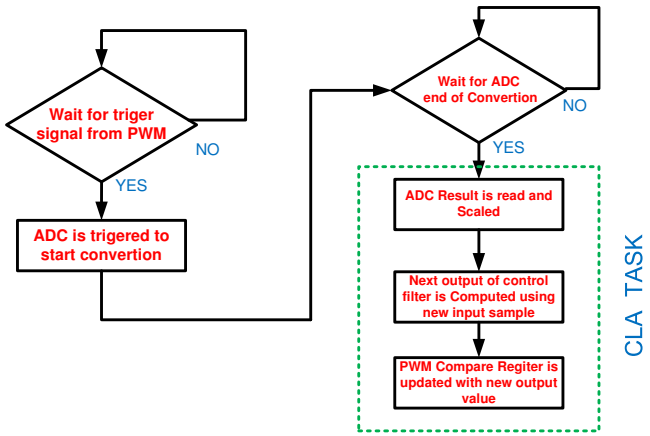

Figure 5.5: CLA task control flow diagram

- Feedback is read using ADC and scaled accordingly
- A new value of feedback is fed to the control filter as a new input sample and a new value of output is computed depending upon internal filter architecture
- A new value of output is scaled with period and written to PWM duty cycle register

We are controlling four SMPS stages and there are four of these loops executing on CLA as shown in figure 5.1. Task 1, 2, 3, 4 are used to implement control loop for boost, buck-boost, buck and SEPIC converters, respectively. Time division multiplexing is used to execute four tasks on a single CLA resource. Let us first look in detail on a single control loop. Task 8 is used for initialization of the buffer structure used by control filters. It will only execute one during initialization.

5.5.1 Control Loop Architecture

The detailed architecture of the control loop is presented in this section. First details of different components will be presented, and later limitations of this will be discussed, followed by improvements and results.

Acquiring Feedback from ADC

The first step is to read a feedback value from the ADC result register. The CLA can directly access the ADC result register. Later this value is scaled; this scaling is dependent on circuit parameters in the feedback path. The firmware for this is integrated into the control filter to decrease instruction count.

Control Filter

2nd and 3rd order control filters from DPL are used in our implementation. These filters can be interfaced in the code by available pointers. Both these filter are actually infinite impulse response (IIR) filters. Both filters have identical interfaces but the internal structure is different due to different filter orders. Data buffers and coefficient structures will have different depth depending upon order of filter. The interface for 3-pole 3-zero filter is shown in figure 5.6. The internal structure of this filter is also presented in the figure.

The 2nd order version of this filter takes 31 clock cycle and the 3rd order filter takes 38 clock cycles.

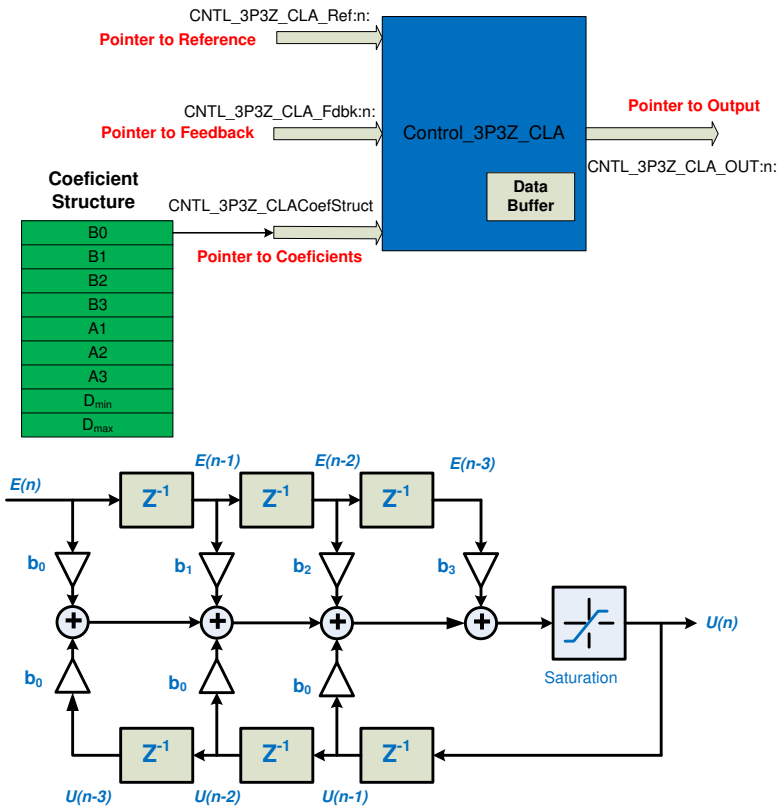

Figure 5.6: 3-pole 3-zero filter interfacing in control loop

PWM Update

The output of the control filter is written to PWM. But this value needs to be scaled according to the PWM period. Moreover, since we are using the high resolution feature in the PWM module, this value should be scaled accordingly. A PWM update driver from DPL is used for this purpose. The interface and the internal architecture is presented in figure 5.7. This driver consumes 12 clock cycles.

5.5.2 System Architecture

The above mentioned components are combined to implement control loops. A few of the necessary steps are listed below. The system implementation is depicted in figure 5.8.

- The feedback pointer is directed to the ADC result register corresponding to feedback

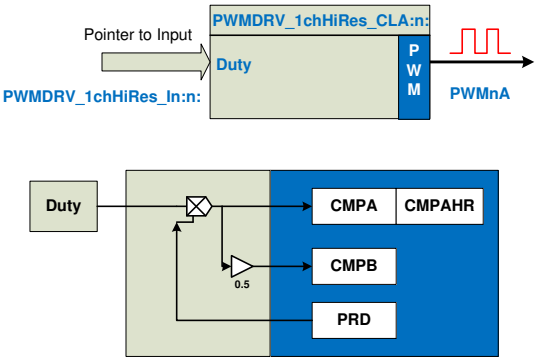

Figure 5.7: High resolution PWM driver

- A variable is initialized for reference. The reference is passed on from user. The C28x CPU will read this value from the watch window and update it in CPU to CLA message RAM for control loop
- The coefficient structure is initialized and placed in CPU to CLA message RAM. This structure will hold filter coefficients
- A variable is initialized for keeping duty cycle value. This variable will serve as output to control filter and input to PWM driver. Note that the address of this variable is passed on to the control filter's output pointer and PWM driver input pointer. This variable must reside in CLA to CPU RAM as this RAM be written by CLA and being used as data RAM for CLA

The control loop with 2nd order control filter will consume 45 cycles and 3rd order control loop will consume 52 clock cycles. These values are found by adding the cycle count for control filter and PWM update; two clock cycles for CLA task start are also added. The maximum limit on switching frequency can be calculated by using equation 5.1.

$$F_S = \frac{F_{clock}}{Cycles} \tag{5.1}$$

In a system with three 2nd order control loops and one 3rd order control loops, the total cycle count will be 187 clock cycles. The upper limit on switching frequency can be found from this value; considering the 60 MHz clock, the upper limit will be 320 kHz. But some modifications are required to improve other aspects of system. That will add some further instructions to the control law.

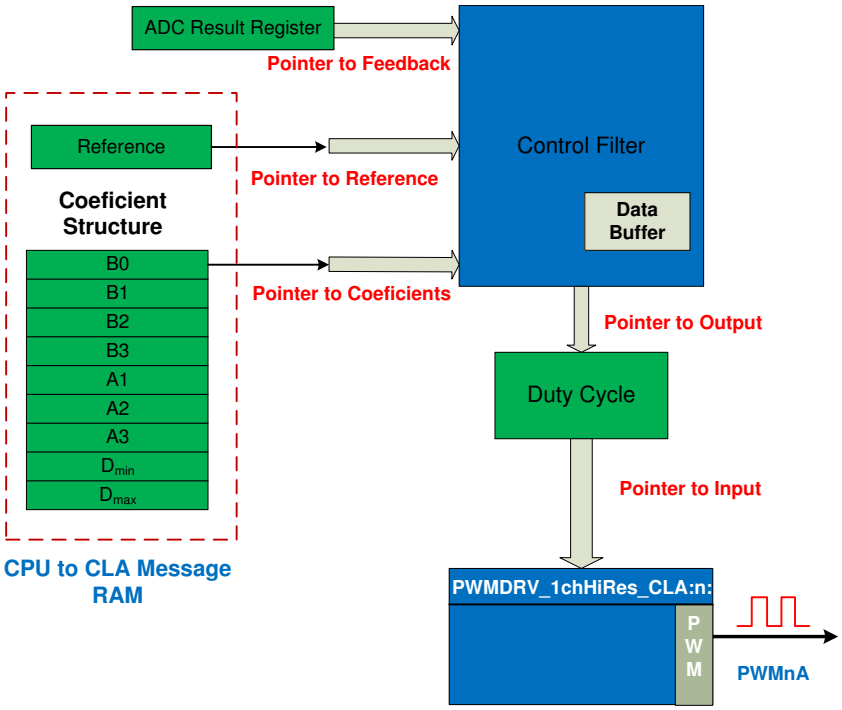

Figure 5.8: Control loop system implementation

5.5.3 Improvements in Control Loop Architecture

Two modifications in the above implementation are required:

Offset Cancelation

The feedback amplifier used to amplify the current feedback has a small offset. The accuracy of current through LEDs is affected because of this offset. This offset is needed to be removed in software. One way to solve this is to subtract the offset value from feedback every time the control law is computed. The value of the offset can be measured during system initialization. A new variable is initialized in CPU to CLA message RAM. The control law is modified to remove this offset. The value is read from the variable containing the offset value and subtracted from the feedback value. The cycle count for the control filter is increased by three due to this modification.

Optimization of Control Law

Drivers used from DPL are very general in nature. Several modifications can be made:

- One of the limiting factors is the number of memory accesses. One way to minimize this is by reducing the number of pointers. We propose to integrate all the variables into one structure. Only one pointer is accessed for the whole structure and then variables can be accessed by incrementing pointer. The sequence of instructions will be changed to make it consistent with the newly developed variable structure.

- The memory access to the variable containing duty cycle value can be saved by integrating two drivers. Moreover, some redundant instructions in PWM driver were removed. There are some NOP (no operation) instructions in the code to avoid data hazards. The instructions can be better adjusted in the integrated driver.

- Instructions are rearranged to decrease NOP count and consequently overall instruction count.

Modified Implementation

The above mentioned modifications are made. The first modification will increase cycle count while the second one will decrease cycle count. The modified driver is presented in appendix sections B.3. The new system implementation is presented in figure 5.9. The new integrated driver consumes 35 clock cycles for the 2nd order driver and 41 clock cycles for the 3rd order driver. The total cycles for four SMPS stages will be 154 and the upper limit for switching frequency in this case will be 388 kHz. In short, this modification was useful in increasing the switching frequency.

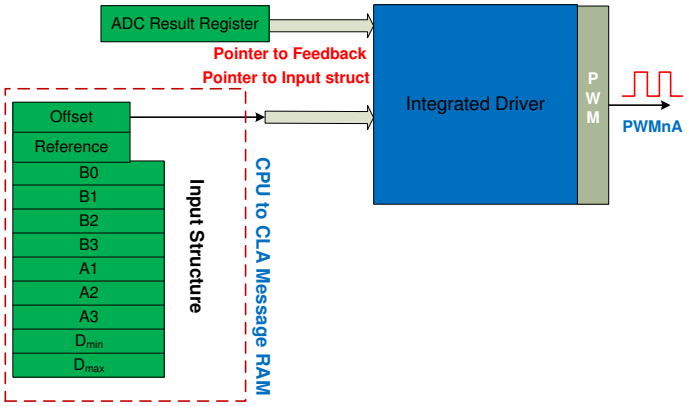

Figure 5.9: Modified control loop system implementation

5.5.4 Control Loop Triggering

The CLA tasks can be triggered by ADC end of conversion, PWM module or by software. Due to limitations on sampling instants, the control law will be triggered when ADC will complete conversion. The ADC start of conversion is in turn triggered by the PWM module. Since this sampling instant is in middle of PWM on cycle, this is configured in the initialization of the PWM module. Task 1,2,3,4 are triggered using the above setup. Task 8 consists of initialization code to be run by the CLA. So task 8 is triggered by user in software once on system startup. A detailed time line diagram of this is presented in figure 5.10.

$\triangle$X represents the time from start of conversion to PWM update by control loop. There are two versions of control filter as described in earlier sections. Let us denote the time for the 2nd order control loop with $\triangle$X1 and the 3rd order control loop with $\triangle$X2. Both these notations will be used later. $\triangle$OVP gives the time from start of conversion to computation of output by the CLA task.

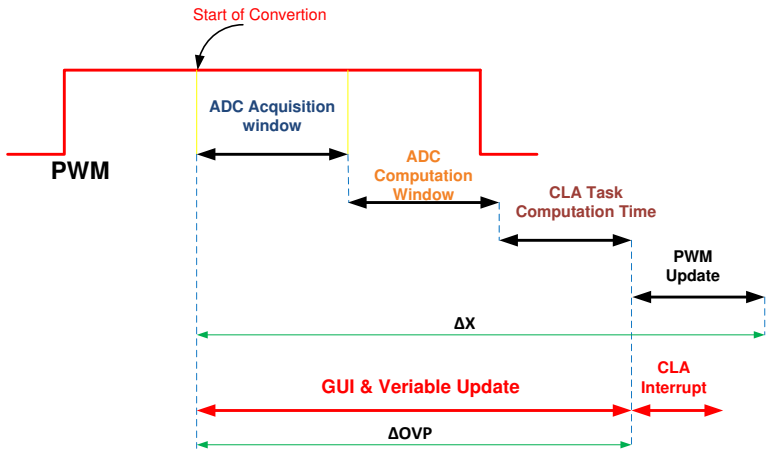

Figure 5.10: Computation sequence of control loop

Since four SMPS stages are controlled with one CLA, time sharing is used to allocated 1/4 of bandwidth to each control loop. Moreover, due to requirements on sampling instants discussed in section 5.3, this time sharing is desired because of single shared ADC and CLA resource. Consequently, the PWM waveforms controlling these SMPS are synchronized and phase shifted by 90 degrees. This scenario is shown in figure 5.11.

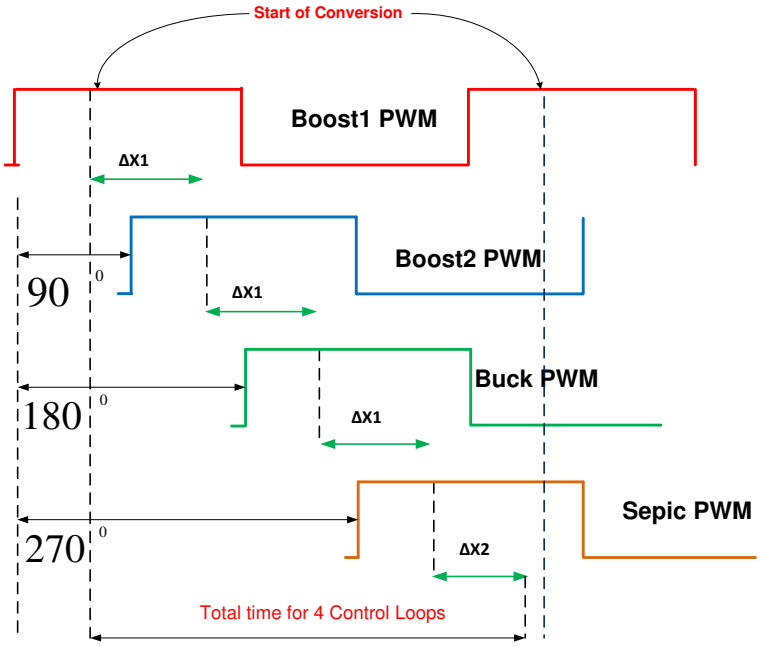

Figure 5.11: Time line diagram of four control loops running on CLA

5.6 C28x Code

C28x, the main CPU, is assigned to monitoring system variables and diagnostics. Important system parameters need to be monitored. This does not only have considerable importance in development phase, but also in final product. Considering a final product, some of these parameters are periodically logged in ECU for later analysis in case of failure. There also exist a number of input parameters that are needed to be passed to system by user. Some of them are used in control loop, i.e., reference. Some will be used by diagnostic functions. In this design phase we used watch windows functionality available in CCS as GUI. The firmware for this is organized in a number of nested state machines. There is one central state machine which will continuously check triggering conditions for nested state machines. This is shown in figure 5.12

State machine A, B, C is triggered by overflow of timer 1, 2, 3, respectively. Dimming state machine will be triggered by a VTimer0[0] reaching user defined value, which is a software timer based on timer 1 overflow. A number of functions exist inside each loop that are allowed to execute one by one using a state pointer concept. The detailed implementation is presented in Main.c in appendix B.1. Function-updating input parameters are organized in different loops based

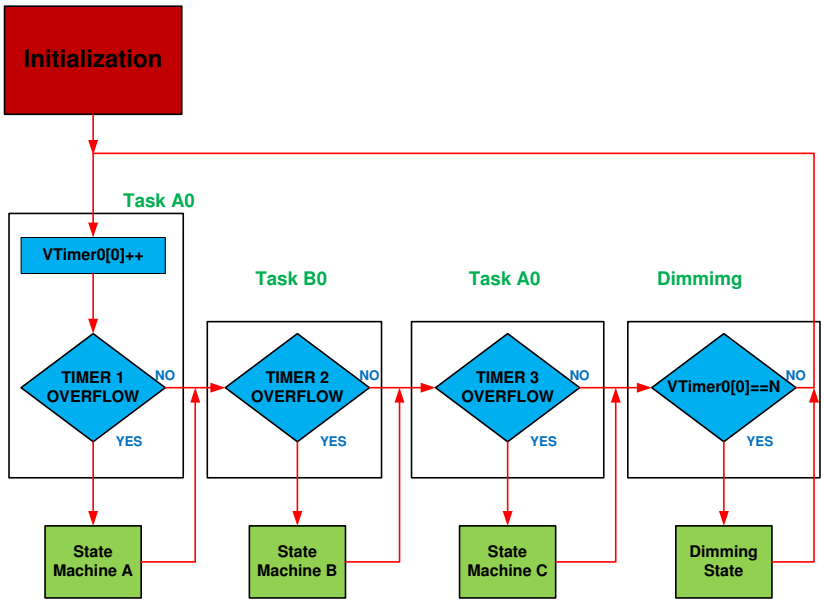

Figure 5.12: C28x software architecture overview

on their update requirement and loop repetition frequency. Using the right loop for a certain parameter is important, because of bandwidth limitations; bandwidth is a valuable resource in such MCU implementations. The details of each state is presented in subsequent sections.

A State Machine

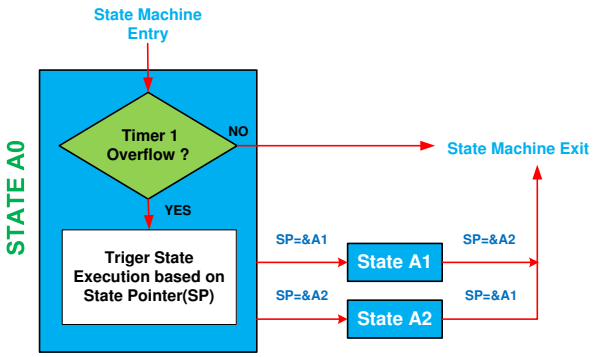

Figure 5.13: State Machine A

State machine A is based on timer 1 which has a period of 1 ms. One of the states is executed whenever this state machine is triggered. Since there are two tasks

in state machine A, the internal state will have a periodicity of 500 Hz. This is graphically presented in figure 5.13. The actions performed in each task are listed below.

- State A1: First, the global power enable input from user is monitored and the main power FET is controlled accordingly. Second, the voltage reference variables for over-voltage protection are updated.
- State A2: This clears the current reference variables in case of trip zone events. Trip zones are activated automatically by the over-voltage function in case over-voltage condition is detected at any of converter.

B State Machine

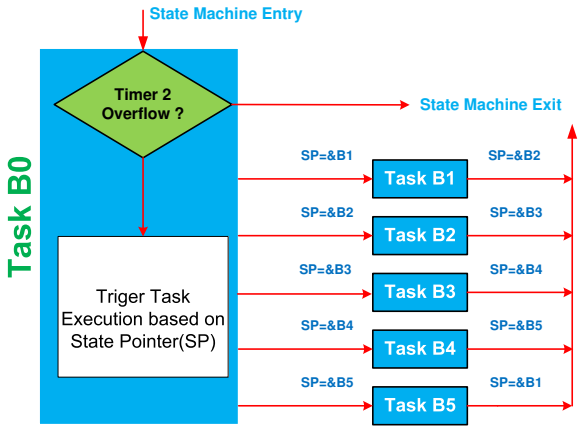

Figure 5.14: State Machine B

State machine B is based on timer 2 which has period of 2 ms; the same as for state machine A. Since there are five tasks in state machine 5, the internal states will have periodicity of 100 Hz. This is graphically presented in figure 5.13.

- State B1 updates the values of input current and voltage to GUI
- State B2 performs two function
  - First, values of SEPIC output current and voltage are updated on GUI
  - Secondly, output current reference value is updated to CLA from GUI as user input
- State B3 performs same functions as state B2 for Boost1 converter

- State B4 performs same functions as state B2 for Boost2 converter
- State B5 performs same functions as state B2 for Buck converter

C State Machine

State machine C is based on timer 3, which has a period of 200 ms. There are two tasks in state machine C, so the internal states will have a periodicity of 2.5 Hz. This is graphically presented in figure 5.13.

- State C1 is just used to blink LED on the control stick to show execution of code
- State C2 is used to clear PWM trip zones in response of user input. Trip zones are required to be cleared to resume normal operation after a shut down in response of an over-voltage condition.

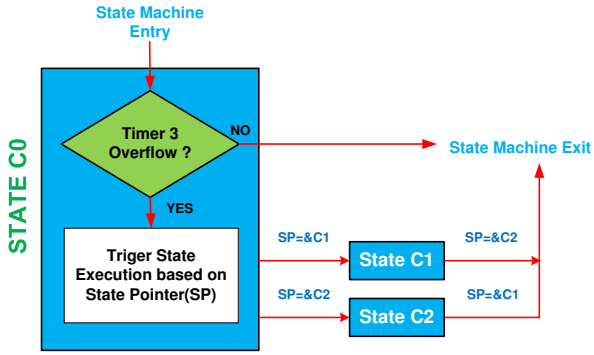

Figure 5.15: State Machine C

Dimming State

Dimming functions are driven by software timer VTimer0[0], which in turn is driven by timer 1. 50 ticks are counted to trigger dimming. That means repetition frequency will be 20 Hz. This state is employed for two purposes:

- It is used to enable or disable ISR that is handling dimming functionality based on user input
- The dimming duty cycle is updated from GUI to PWM registers

The architecture of the dimming task is depicted in figure 5.16. Dimming is only implemented for three converters, i.e., boost, buck-boost, and SEPIC, because of availability of only three free PWM modules. Other considerations for dimming are discussed in section 5.7.

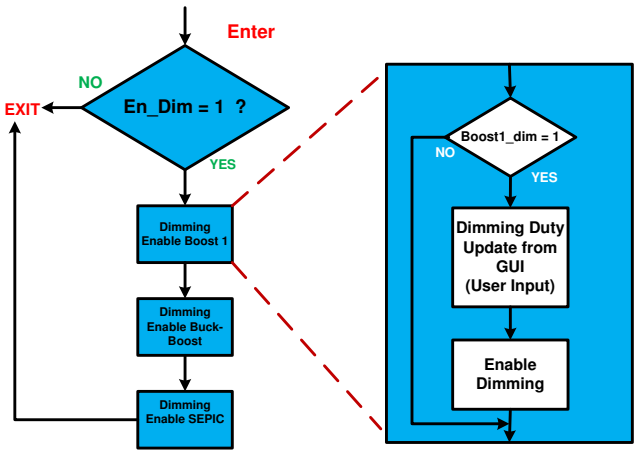

Figure 5.16: Enable dimming task architecture

5.7 PWM Dimming

An important characteristic of LED light is that current is converted into light. Decreasing current through LEDs will decrease light intensity in headlights. However there are two limitation with this technique. A specified wavelength emitted by an LED is at a certain current and the wavelength will change a little if the current is higher or lower than the specified current [10]. That means a change in average current through the LEDs will change the color, which is not desirable in the case of automotive headlights and the associated commercial regulations. Since SMPS are used to control the LED automotive headlights, the current in the LEDs can be controlled by changing the duty cycle. However, at lower duty cycles these SMPS operate in discontinuous conduction mode (DCM) instead of continuous conduction mode (CCM). We want our converters to operate in CCM as it is part of specifications. So the concept of PWM dimming will be employed. The full range of dimming can be used by using PWM dimming while operating in CCM. To implement PWM dimming in this configuration, a FET is introduced in the circuit along with the grounding path. Figure 5.17 shows this configuration.

The dimming PWM signal will be derived at much lower frequency as compared to the main PWM. One important point to be noted is that the main PWM should be turned off during the off time of dimming PWM to avoid voltage and current spikes, which could be dangerous for the hardware platform. Moreover, based on same arguments, both PWM edges should be synchronized. The output tripping feature available in ePWM modules in TMS32F28035 is used for this implementation. Tripping is a protection feature in which the ePWM output pin can be pulled to low, high or high impedance state. The trip zone can be trig-

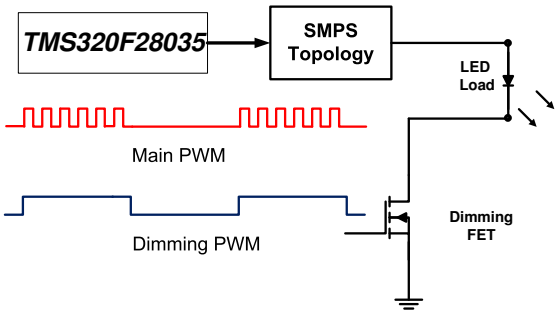

Figure 5.17: Circuit Description of Dimming

gered from software or from hardware using external GPIO pin. Please note that when the trip zone is triggered, the GPIO pin corresponding to a PWM module is disconnected from the PWM module while the module still keeps on running internally. The trip zone will be triggered for the main PWM, when the dimming FET is switched off by dimming PWM signal. Similarly, the trip zone for the main PWM is cleared when the dimming FET is turned on. There are two important considerations related to implementation:

5.7.1 Limitation I

A high level of synchronization is required between both PWM modules. This is achieved by using PWM interrupts. But there is a issue of delay between triggering and execution of Interrupt Service Routine (ISR). This is handled by triggering ISR well before the PWM edge. This is possible by the help of a second compare register available in the ePWM Module. This behavior can be clearly understood by figure5.18.

One could argue that the main PWM will turn off or on before dimming PWM. This situation will not create any voltage or current spikes. The dimming task is given precedence over other CPU tasks by using ISR for implementation. But there are other ISRs which also have critical importance, like the one discussed in section 5.8.1, which could alter this synchronization between ISR execution and PWM edge. Relative priorities between ISRs is a trade off.

Before moving further, it is good to summarize the actions taken in ISR:

- Trip the main PWM when dimming PWM goes low and vice versa
- Interrupt source has to change from rising edge to falling edge or vice versa in each ISR call
- Compare B register in PWM module has to change so that we always trigger the interrupt well before the PWM edge

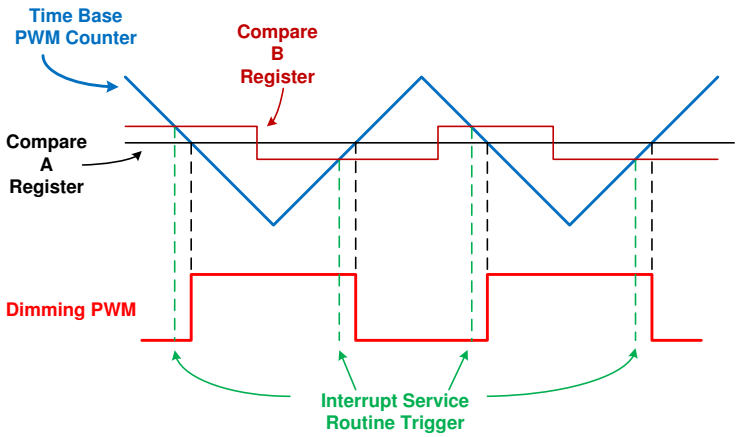

Figure 5.18: Triggering of ISR handling dimming function

5.7.2 Limitation II

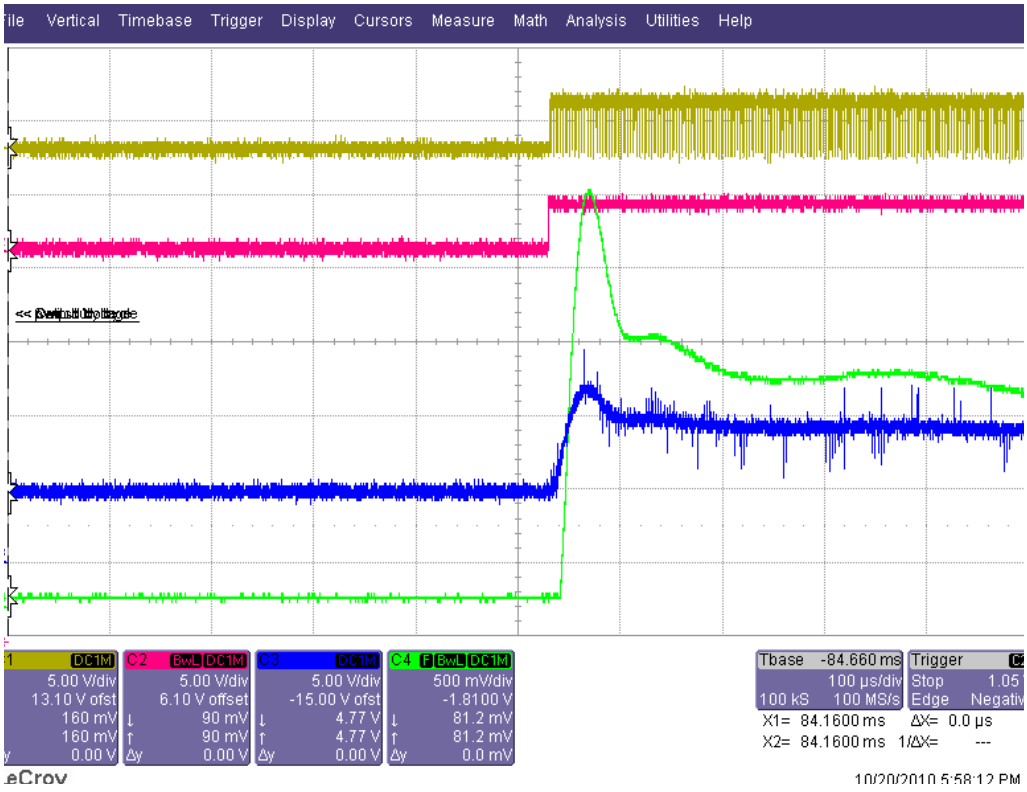

Figure 5.19: Screen shot of dimming

As discussed before, the PWM module is running while trip zone is activated. That means the ADC is triggered periodically by PWM, which in turn will trigger the CLA task. In short, the control loop is compensating during the off cycle of dimming PWM. During the off cycle of dimming PWM, the current through LEDs will fall to zero, consequently feedback will fall to zero. The CLA will compensate this by increasing the PWM duty cycle until it reaches the maximum allowable value. Remember that the PWM module's output is disconnected from GPIO pin and the CLA is only updating the PWM compare register. As soon as this trip zone is cleared by ISR in the on cycle of dimming PWM, this PWM with maximum duty cycle is applied to the SMPS stage, which will respond by increasing its output. So there will be a large current rise. A screen shot from real measurement is presented in figure 5.19. The curve in green and blue show current and voltage respectively.

This high current will eventually be compensated by the control loop but this has two drawbacks: First, this needs considerable time and, secondly, such a high current can decrease the lifetime of LEDs and other components on the EVM. A number of solutions for this are implemented and tested until we came across the most suitable one. All of them are discussed here:

- The first option is to let PWM and CLA run as they are. There will be high rush current on the rising edge of the dimming PWM so this option is not feasible.

- The next obvious option is to try to restrict the duty cycle. First the current reference to the CLA is switched to zero during the off time of the dimming PWM and back to the original in on time. The duty cycle will drop during the off time and has to rise during the on time; this will require an extremely fast compensation.

- Next, the maximum duty cycle was fixed to a pre-conceived value so that when it is increased by the CLA, it will be ceiled to that value and could not increase further. This implementation performed better than the above proposals but it has its own drawbacks. The pre-conceived maximum duty cycle suitable for best performance is dependent on the DC operating point.

- The conclusion that can be improvised from the previous implementation is that the duty cycle should be fixed to the value just before turning off the dimming PWM as this will have fastest response as a head start will be given to the compensator as well. This technique is implementated and tested to give the best response.

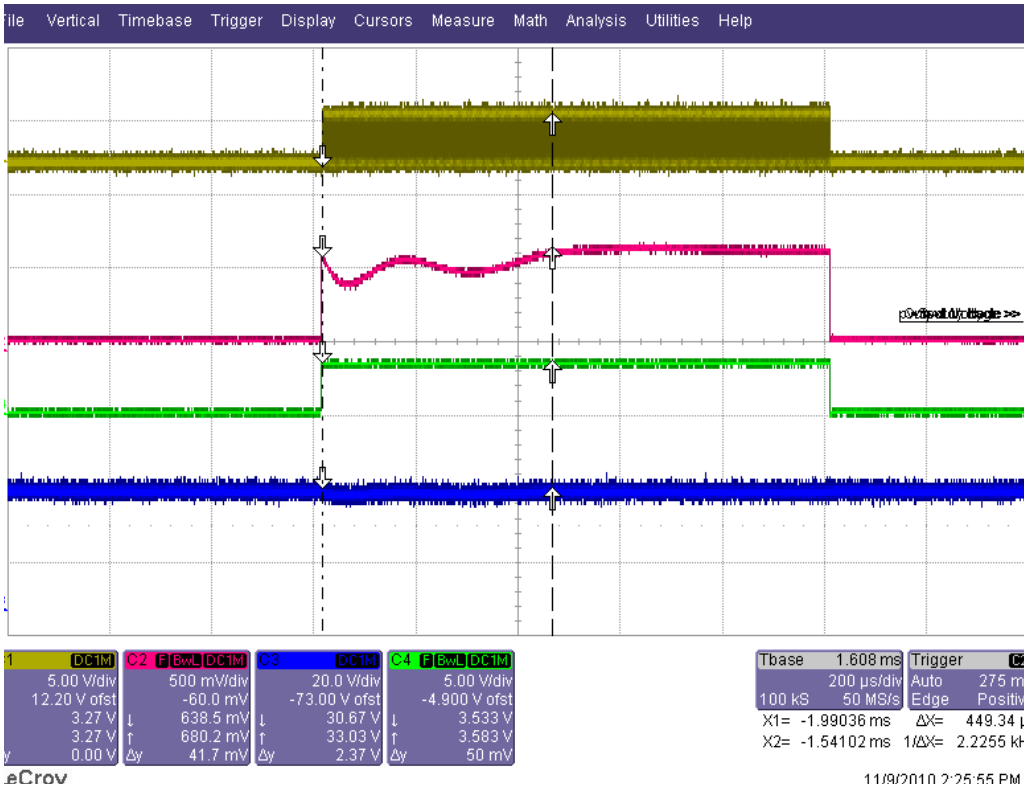

Figure 5.20: Screen shot of dimming after improvements

Just to summarize the above discussion, the last technique is the best among all to implement PWM dimming. There are a number of possible alternatives to implement this, but the simplest one is chosen, that is, by disabling the corresponding CLA task. This is done by disabling the corresponding ADC interrupt, which is used for triggering that CLA task. Other CLA tasks are executing and will not be affected. A screen shot using this technique is shown in figure 5.20. The output current and voltage is shown in red and blue, respectively. This measurement is done with a relatively slow compensation; that is why there are some ripples in the start, but it compensates after some time. The time taken by compensation defines the upper limit on the dimming frequency. The source code for this is presented in section B.5

5.8 Diagnostics

The next important part of software implementation is the diagnostics. These functions are important for the protection of the hardware platform. Considerable system bandwidth is however needed to be allocated to these functions. An initial

implementation analysis is done for allocation of bandwidth to different diagnostic functions. Post-implementation analysis is useful in verification and modification of pre-implementation analysis.

5.8.1 Over-Voltage Protection

Over-voltage protection is the most important diagnostic function. An over-voltage condition not handled in the specified time period could result in damage to hardware. The bandwidth for the over-voltage protection (OVP) function is a major trade-off:

- A high repetition rate for over-voltage check will result in suffocation of other CPU tasks.
- Too low repetition rate can expose the EVM to over-voltage condition for long-enough that can result in damaging EVM.

A detailed time-line diagram of the control loop implementation is presented in figure 5.10. ADC acquisition and computation times are constant for a specific configuration of the project. The execution time for the CLA task is dependent on the number of instructions, which are in turn dependent on the complexity of the compensator.

First Draft Implementation

In the first implementation the over-voltage check is made in every CLA task ISR. The repetition rate for OVP is the same as the switching frequency. This implementation is really fast, but consumes a lot of bandwidth. The details of this implementation are shown in figure 5.21.

Second Draft Implementation

The initial post-implementation analysis of OVP showed that the rate of increase of voltage is low enough to allow the repetition rate of OVP to be reduced to free system bandwidth. Secondly, every ISR takes considerable clock cycles for context saving. So we propose that instead of checking over-voltage condition for each topology in its own ISR, an over-voltage condition for all channels can be checked in one function. The ISR of the last CLA task in the whole time-line is used to implement this global OVP function. The repetition rate of OVP is the same as before. A considerable number of clock cycles initially lost in context saving were saved by combining four ISRs in one. This second implementation is presented in figure 5.22

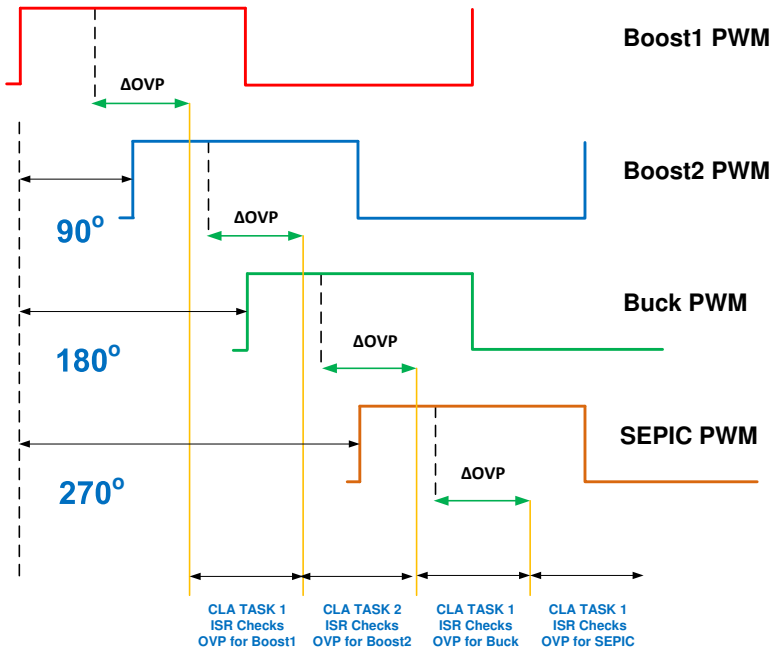

Figure 5.21: Over-voltage protection implementation I

Timing Analysis

A screen-shot from real measurements is shown in figure 5.23. A time analysis is carried out for the OVP task based on this. The purpose of this is to find lower limits on repetition rate for OVP. The delay between two consecutive OVP checks in the first implementation was 3μ s. The measurement showed that the maximum rate at which voltage can increase under over-voltage condition is 17 kV/s. Note that this value is dependent on the response time of compensation and independent of OVP implementation. First, the voltage increase between two consecutive over-voltage checks will be calculated

Voltage increase in 1 s:

$$V = 17kV \tag{5.2}$$

Voltage increase in 3 μ s:

$$V = (17 \times 10^3) \times (3 \times 10^{-6}) = 0.051V \tag{5.3}$$

So the voltage increase between two consecutive over-voltage checks is 0.051 V which is a very moderate value. This analysis gives the liberty to decrease the repetition frequency of over-voltage checks to free the bandwidth for other tasks.

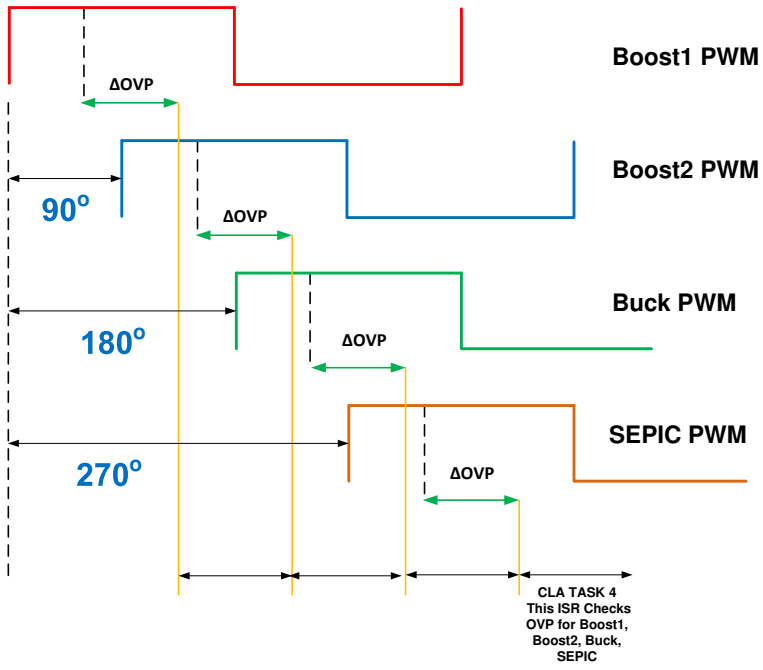

Figure 5.22: Over-voltage protection implementation II

Calculation for a minimum repetition rate for OVP is valuable information and very useful for further improvement in bandwidth allocation.

Consider x V as the maximum voltage that could be allowed to increase between two consecutive over-voltage checks. Then the rate of voltage change is:

$$\triangle S = \frac{1sec}{17 \times 10^3 V} = 5.88 \times 10^{-5} \frac{sec}{V} \tag{5.4}$$

For voltage x V, value will be

$$\triangle S_x = x \times 5.88 \times 10^{-5} sec \tag{5.5}$$

The lower limit on repetition rate depends on the variable x. A power converter is designed to withstand certain maximum voltage. Moreover, circuit have a tolerance limit i.e. circuit can avoid catastrophic failure for certain voltage above maximum operating voltage. However, circuit can only with stand this voltage for certain time. When using over voltage protection as periodic software check. The

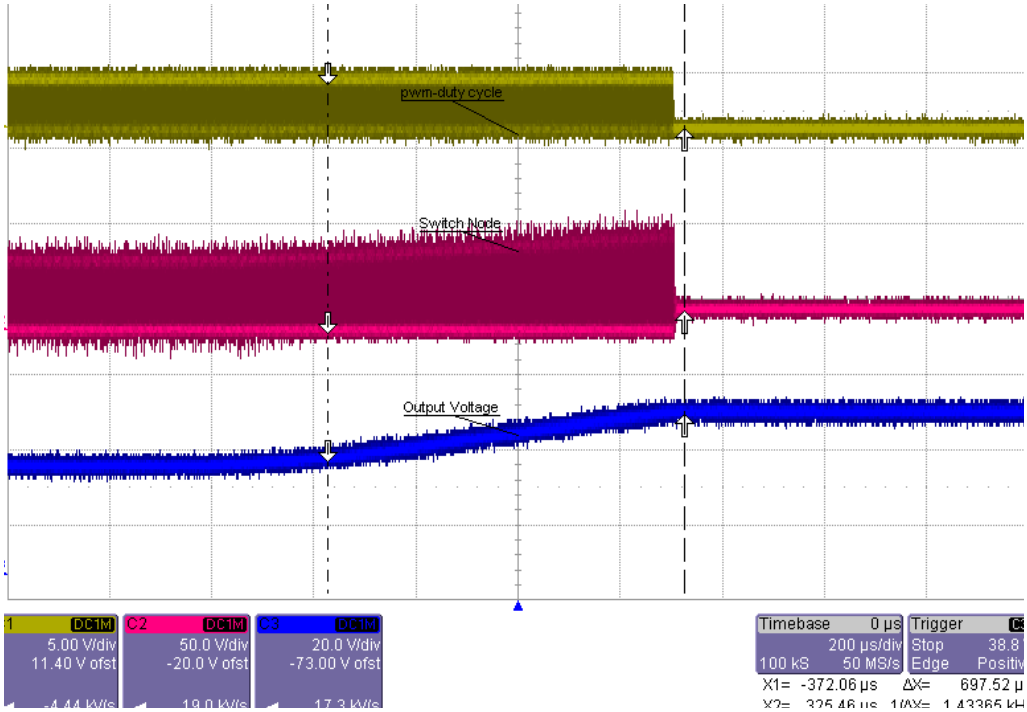

Figure 5.23: OVP screen shot for boost SMPS

repetition frequency of this check determine the response time of system. So voltage above maximum operating voltage becomes an important factor. x represents the voltage above maximum operating voltage that could be allowed to increase to avoid any component failure between two over voltage checks.

Chapter 6

Measurement and Results

The frequency-response analysis is a measure to verifying performance of a system. This analysis exposes system behavior in terms of gain and phase with frequency. Primarily, there are two types of tests. First, the open-loop test; as the name implies, this is carried out by opening the feedback loop. This test provides information on system behavior without effect of compensator. Second, the closed-loop test; this test is done with the feedback loop intact. This test provides information on the combined behavior of compensator and converter. The purpose of closed-loop tests is to compare post-implementation system behavior with the models used during design. This comparison provides some important insights into digital power management tradeoffs. Both types of measurements will be carried out on our system and details about measurement setup is discussed in detail in subsequent sections.

6.1 Measurement Equipment

The Venable Windows software in combination with the supported Frequency Response Analyzers (FRA) is a complete frequency-response modeling and measurement system [5] that is used for measurements in this project. The hardware portion consists of the 3235 FRA, which is used for making measurements of gain, phase, and voltage versus frequency, and various accessories for coupling the FRA to the electrical system under test. The software portion runs on any personal computer. The 3235 FRAs are controlled through a National Instruments GPIB board. This software also contains a simple SPICE-like modeling program for modeling the AC frequency response of circuits. Implementation model results and test results are in the same format and can be displayed simultaneously for easy comparison. Compensation amplifier synthesis software lets the user achieve the exact feedback loop bandwidth and phase margin. Math software

allows any kind of mathematical function on any one or two transfer functions. Graph types supported are voltage versus frequency (log-log), gain and phase versus frequency (semi-log), reactance versus frequency (log-log with lines for constant capacitance and inductance), and Nyquist (log outside of the unity gain circle and linear inside the unity gain circle). The FRA has a simple interface that consists of an oscillator, channel 1 and channel 2. The oscillator provides the AC signal and the DC bias. Channel 1 input serves as primary input, while channel 2 input serves as reference input [5].

6.2 Open-Loop Measurements

The open-loop transfer function refers to the gain from the output of the error amplifier to the output of the system. This gain block typically has a fixed low-frequency gain. The high-frequency gain falls off at a -1 (-20 dB/decade) or -2 (-40 dB/decade) slope depending on the characteristics of the circuit. Because the gain at low frequency (including DC) is fixed, it is possible to use a DC voltage to bias the operating point to achieve the desired system output. By superimposing a small AC voltage on the DC bias voltage, the operating point of the modulator can be varied. The transfer function of this gain block can then be measured by connecting frequency selective voltmeters (the inputs of the FRA) to the input and output of the circuit and sweeping the modulation frequency across the desired frequency range. The output of the Venable 3235 FRAs are designed to deliver DC and swept frequency AC voltage simultaneously. The inputs are designed to measure voltage at the frequency of the output and reject all other frequencies and DC [5].

The first step is the setting up of circuit to be controlled by FRA that is done by breaking up the feedback loop and connecting the oscillator (channel 1, channel 2) at appropriate points. There are three ways these terminals can be connected depending on the error amplifier. Please note that none of these methods is suitable for a digital controller. These are just mentioned to give a preview of the problem.

6.2.1 Method I

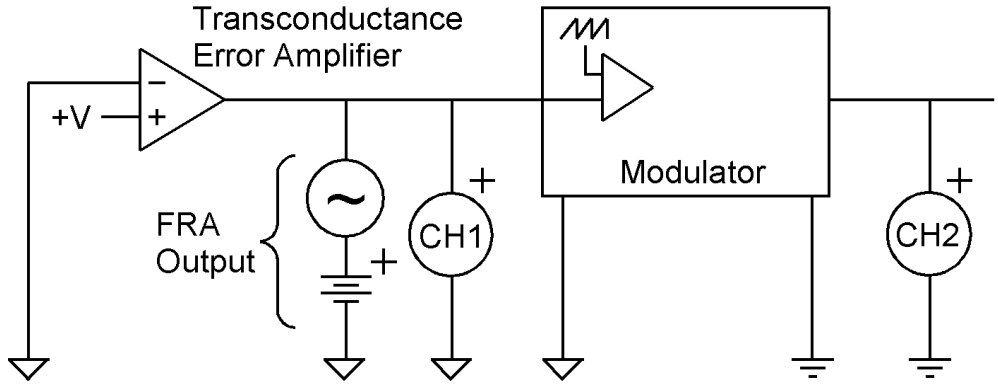

Figure 6.1: Open-loop measurement circuit setup I [5]

If the error amplifier is a high output impedance transconductance amplifier, the output can be biased high and the output of the FRA used directly to control the operating point [5]. This setup is shown in figure 6.1.

6.2.2 Method II

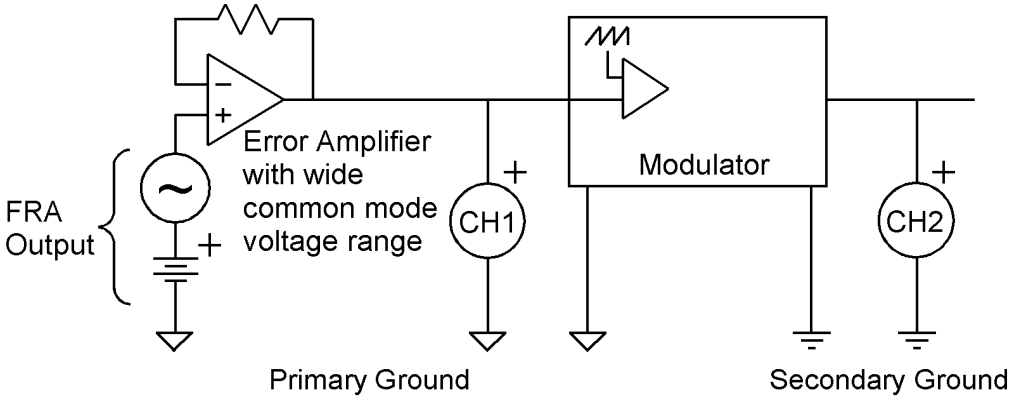

Figure 6.2: Open-loop measurement circuit setup II [5]

If the error amplifier has a conventional low-impedance output but has an input common-mode range at least equal to the voltage swing needed in the output to control the system, the error amplifier can be wired as a buffer follower. This setup is shown in figure 6.2.

6.2.3 Method III

If the error amplifier has a conventional low-impedance output and a relatively narrow input common-mode voltage range that does not encompass the entire output voltage swing required, the error amplifier can be wired as a gain stage and there is complete freedom of operating point. If in doubt, this third method will work in any situation. This third method is shown in figure 6.3.

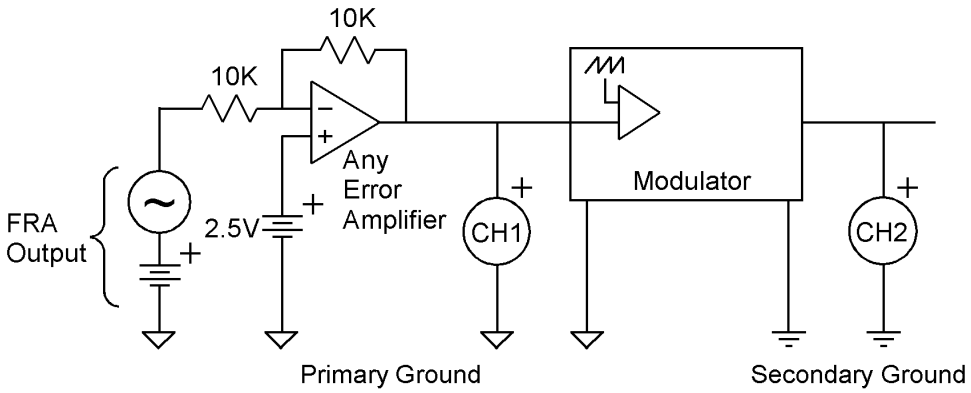

Figure 6.3: Open-loop measurement circuit setup III [5]

6.2.4 Digital Controller Measurement Setup

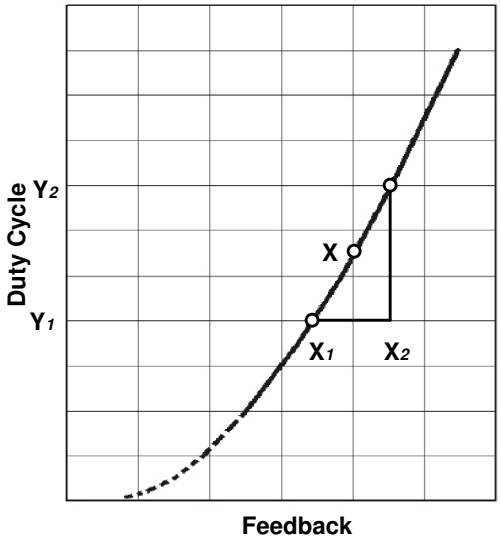

Figure 6.4: Open loop gain for buck converter

The measurement setups mentioned are not valid for a digital controller, as they rely on injecting a signal either after error amplifier or before it. In the case of a digital controller, the error amplifier is hidden in the software implementation so there is no connection to it. The scenario in which noise is injected before the error amplifier and after the feedback is valid. Furthermore channel 1 is always connected after the error amplifier which is not possible. A modification is required to get a new test setup. Since we only have a feedback pin available as an interface to the digital controller, reference values are also provided by software. This can be solved by developing a new firmware component that can translate feedback value to duty cycles based on the compensator's characteristic behavior. The operating point is also needed to be adjusted in software. A curve between feedback and duty cycle is plotted by empirical testing using a closed-loop implementation. Please note that this graph is plotted with only a few points in proximity of the Q point. Two points are chosen before and after the Q point as shown in figure 6.6. Assuming linearity of the graph around the Q point, the feedback to duty cycle translational factor is equal to the slope of the curve. Moreover the operating point is achieved by transept. Since we intend to provide the operating point in software, the duty cycle of a point before the Q point is taken as transept. The slope can be given as

$$Slope = M = \frac{Y_2 - Y_1}{X_2 - X_1} \tag{6.1}$$

The general slope transept equation can be given as 6.2. This is modified to get equation 6.3. The term Y_1 is a duty cycle value and will serve as DC offset. The term $FeedBack - X$ is used rather than only $Feedback$ so that the output can oscillate about the Q point. Firmware is developed to implement this equation. Please note that this technique is applied to all converters to get their values of slope and DC-offset. Consequently, firmware is developed for each converter.

$$y = C + M \times X) \tag{6.2}$$

$$DutyCycle = Y_1 + Slope \times FeedBack - X \tag{6.3}$$

The feedback loop is broken to setup hardware for open-loop measurements. Noise is injected on one side and the response is measured on the other side. Figure 6.5 shows this setup. The loop can be broken either before the feedback amplifier or after it. In both cases, measurement is valid. The oscillator and channel 1 are coupled and connected to the noise injection point. Channel 2 is connected

on the other side so as to measure system response. There is an important consideration in this regard that input and output points lie in close proximity; gain will very low for such setup. This technique for open-loop measurement for digital power controller based converter is improvised but need further analysis. The details of software implementation can be found in C for buck, boost and SEPIC converters. The DC bias needed to operate on desired Q point is provided primarily by software. A small value is also introduced from FRA so as to fine tune the Q point. An AC value of the noise signal is selected so that feedback could not exceed points X and Y, because our translational factor and transept is only valid in this region.

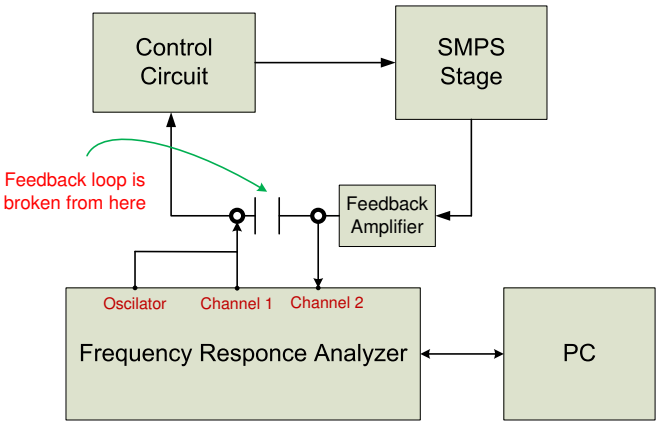

Figure 6.5: Open-loop measurement setup

6.2.5 Measurement Results

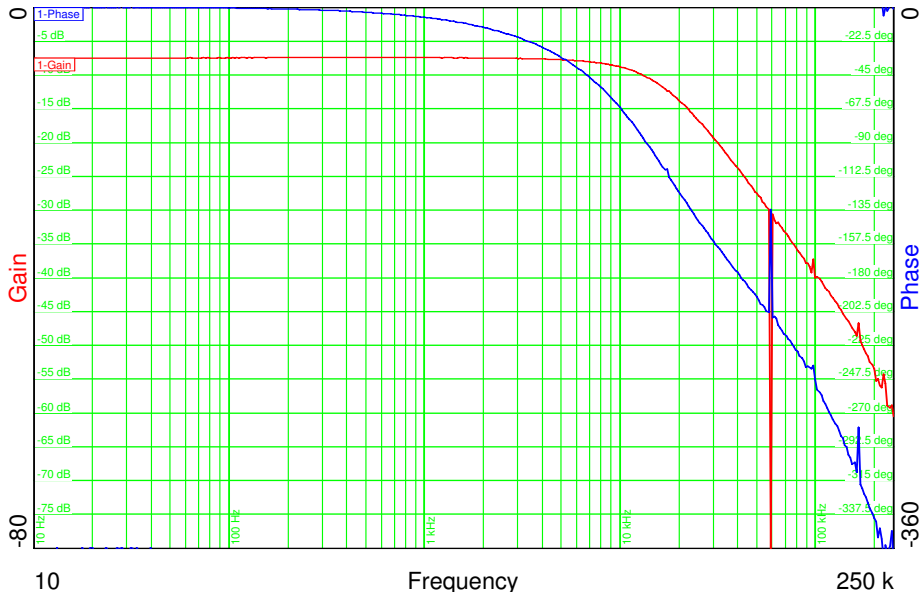

Figure 6.6: Open-loop gain for buck converter

The detailed implementation of firmware for the open-loop test for the buck converter is shown in C. The code for other converters is similar with different values of DC-offset and slope factor. The bode plot of measured open-loop transfer function for buck, boost and SEPIC is shown in figures 6.6, 6.7, 6.8, respectively. Phase and gain scaling with frequency is provided in two separate graphs. The phase response of all the converters is consistent with modeling results with minor deviations. This can be explained by tolerance in components used on the PCB. The gain response differs largely from expected, from the modeling results. This is due to fact that in our setup CH1 and CH2 are connected at the same point. Theoretically, the gain should be zero at low frequencies in our case. However, the low-frequency gain in our case is around -5 dB. This minor deviation can be explained as a shortcoming in the data representation because existing data representation has its roots in analog control measurements. Moreover, this setup for open-loop measurement is new and needs further refinement. In short, development and use of this test gives valuable insight into digital controller performance measurements setups. However, new analysis techniques are needed to fully exploit capabilities of this test.

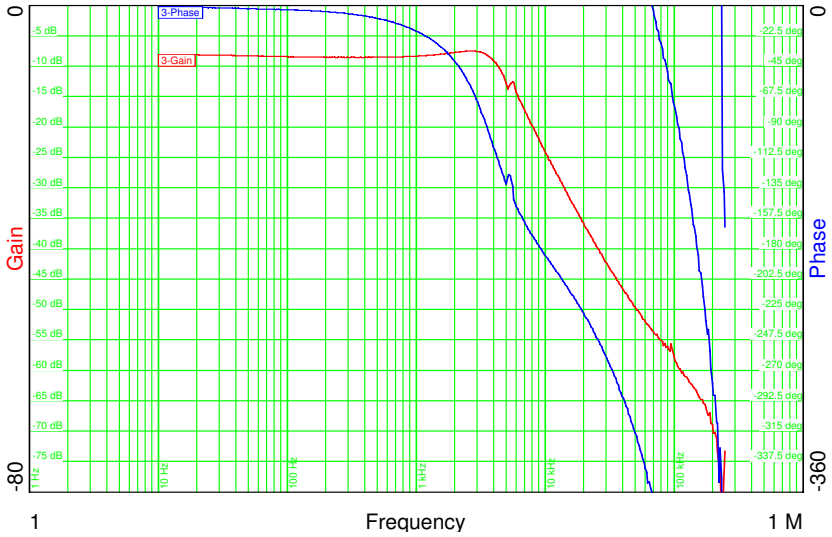

Figure 6.7: Open-loop gain for boost converter

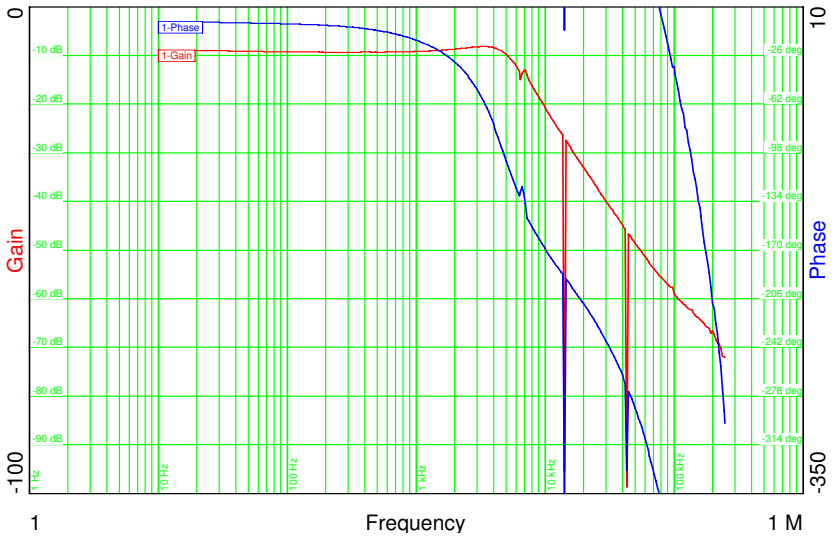

Figure 6.8: Open-loop gain for SEPIC converter

6.3 Closed-Loop Measurement

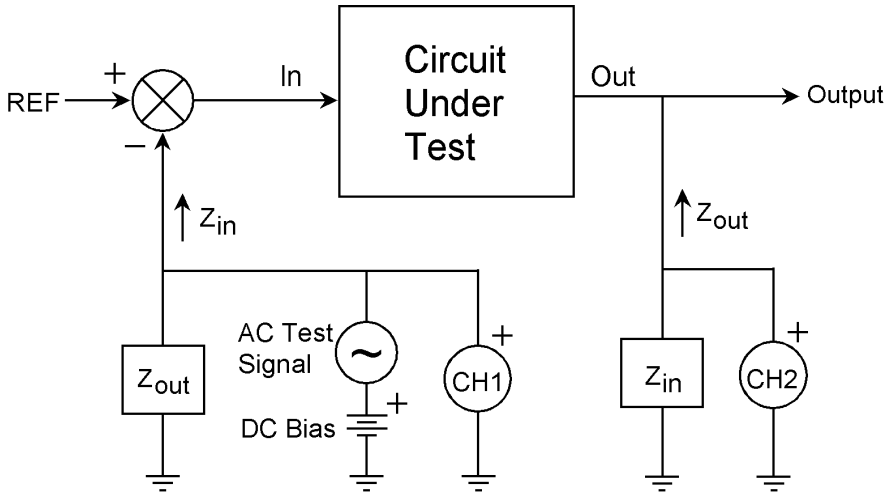

Figure 6.9: Classical method for measuring frequency response [5]

The digital control loop implemented is verified by closed-loop measurements. The classical way to measure a feedback loop transfer function is to break the loop at some point, terminate the input with the output impedance, terminate the output with the input impedance, drive a small AC signal into the input and measure the ratio of the output to the input. In real life, this measurement approach is virtually impossible since the loop gain is usually very high at low frequency and it is difficult to keep the input stable enough to prevent the output from swinging wildly from limit to limit. The classical approach is presented in 6.9 while the Venable approach is presented in figure 6.10.

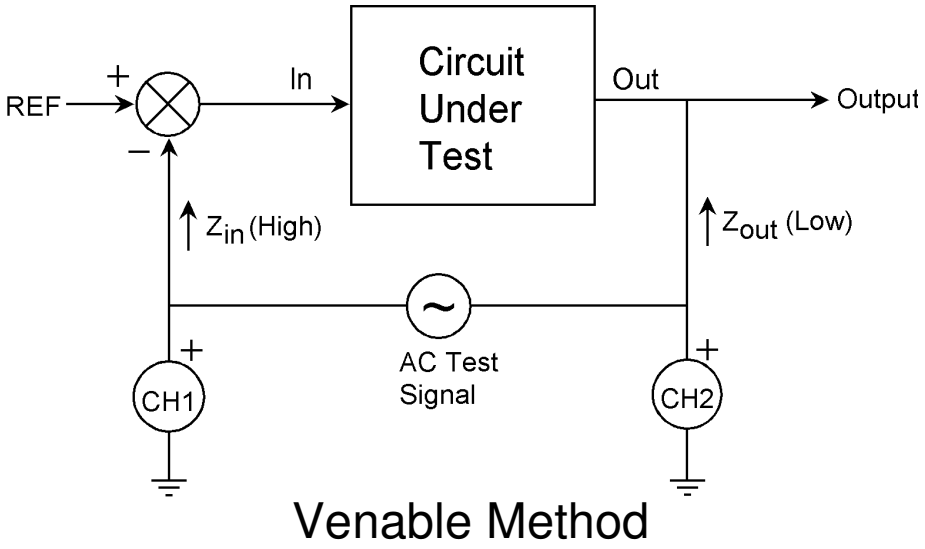

Figure 6.10: Venable method for measuring frequency response [5]

This difficulty in the measurement situation is avoided by finding a place where the loop is confined to a single path (also a requirement in the classical method) and a place where the signal comes from a low-impedance point and drives a high-impedance point. This impedance condition minimizes the error caused by not properly terminating the input and output. We then insert a small resistor into the feedback loop (small compared to the input impedance of the loop). Finally, we connect a floating AC source (the output of a transformer) across the new resistor and drive the primary of the transformer with a sinusoidal voltage source. This converts the resistor into a floating sinusoidal error voltage in series with the feedback loop. This voltage modulates the operating point of the entire circuit [5]. In the context of a digital implementation, this single path is available between the feedback amplifier and the ADC channel. This setup for closed-loop measurement is presented in figure 6.11. Noise will be injected over a resistor in the feedback loop using the injection transformer (BodeBox) as mentioned above.

The feedback loop measurement for boost and buck converter is shown in figure 6.12 and 6.13, respectively. The phase response is consistent with compensator model. Minor deviations are there due to component tolerance as the compensator model is based on an ideal converter model that assumes ideal components. The gain response is different because of the same reasons explained before in section 6.2.5.

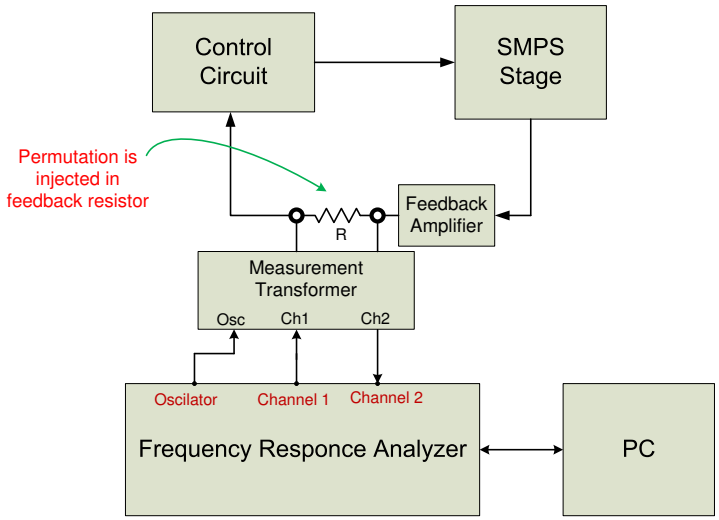

Figure 6.11: Closed-loop measurement setup

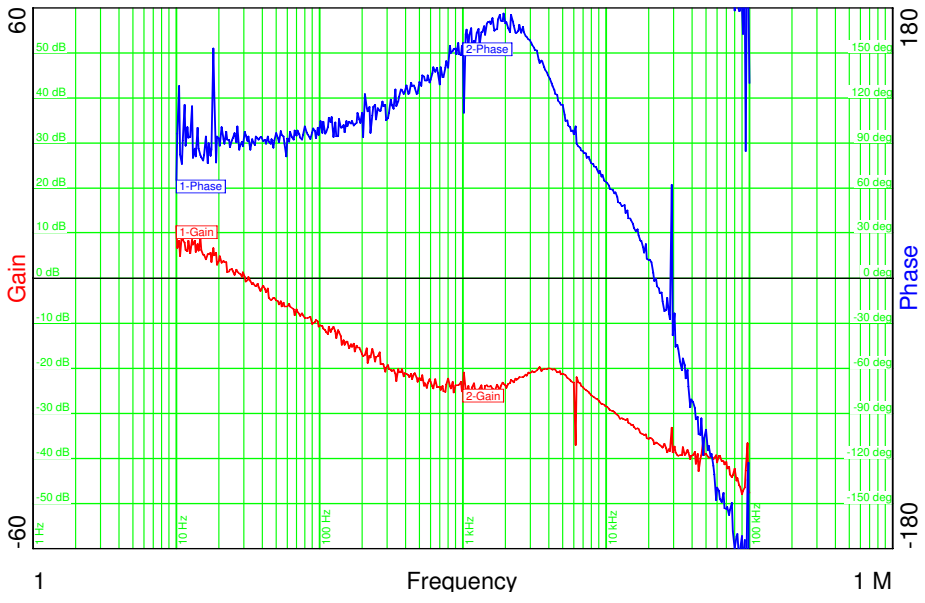

Figure 6.12: Boost feedback loop measurement

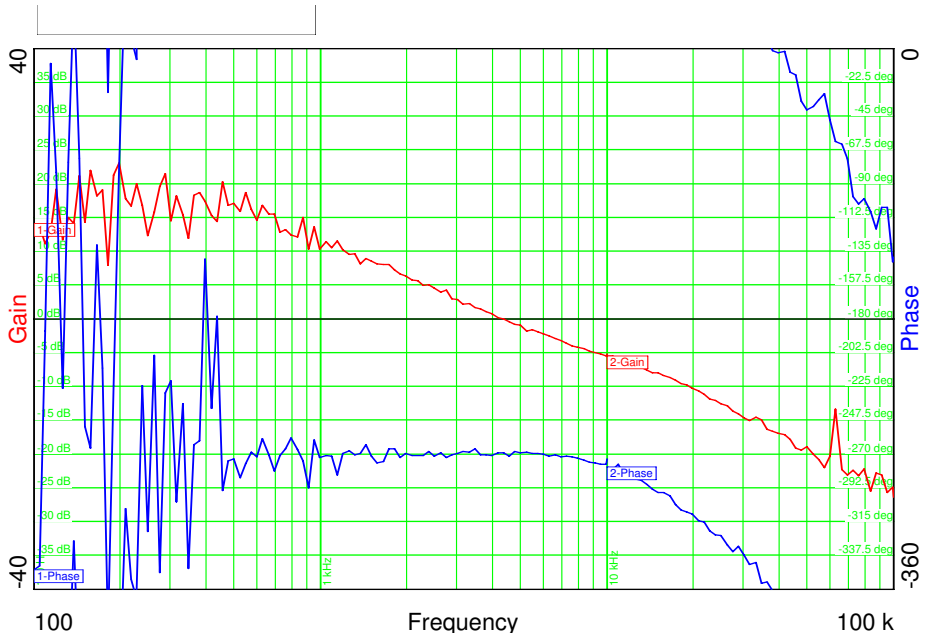

Figure 6.13: Buck feedback loop measurement setup

6.4 Hardware Platform Improvement

There are issues that are exposed after the firmware development on the EVM. Improvements are required in the EVM to improve performance. Moreover, there are some suggestions regarding improvements in the micro-controller. The proposed improvements in EVM are closely concerned with this project. However, they also have a general perspective as well, general to similar applications.

Consequent ADC Triggering

While dealing with real-time applications, it's important to have a deterministic analysis of different software tasks. This will help to have an accurate timing analysis. Moreover, a discrete system implementation is required to sample feedback signal at particular instants as discussed in section 5.3. This adds further importance to deterministic timing analysis. Non-critical signals, i.e., output voltages, are sampled in time slots between sampling of critical signals, i.e., output currents. For example, in the boost converter all the non-critical signals are sampled after sampling output current feedback. These conversions should be complete before the buck-boost converter requests sampling of its current feedback. An ideal scenario would be to start ADC conversion of non-critical signal from end of conversion (EOC) signal of critical one. In general, an EOC should trigger a start of conversion (SOC). That means simply routing EOC signal of one SOC to start another SOC. Unfortunately this is not possible. A technique based on polling and priority is used instead.

Summarizing the above discussion, a functionality of triggering start of conversion of SOC from EOC of another one is needed. This is not available on this TMS320F28035 MCU. This capability will increase the deterministic analysis of real-time applications. From a hardware design's perspective, a few multiplexes are required in the signal routing path. That will not have a significant overhead but yield a far better application analysis.

Inductor and Output Current feedback

In current mode, it is required to sample output current and inductor current values at same time due to reasons discussed in section 5.3. There are two sample and hold circuits in TMS320F28035. But simultaneous sampling is only possible with pair of channel one from each group, i.e., group A and B. That means A0 can only be simultaneously sampled with B0. This implies that the output current feedback should be connected to A0 and the inductor current feedback should be connected to B0 for any converter. Our EVM does not not support this connection as this limitation was not known at the time of board development. It is suggested to

improve EVM to have these connections. However, the MCU can also be modified in future versions to remove this limitation in hardware.

Chapter 7

Conclusion

Voltage-mode control for a converter driving LEDs poses some limitations on digital controllers. First, this technique is an indirect way to control current. The accuracy of the controlled parameter, i.e., the current, is considerably low. Secondly, the converter models used in this technique are complex, requiring a complex compensator of the same order. This fact poses limitations on switching frequency. A limited improvement in switching frequency can be made by optimizing firmware but this technique has some inherent limitations. These converters are inherently non-linear and averaging approximations are applied to develop linear counterparts. These approximations are based on small signal linearity around a large signal operating point (Q point). We used these linear continuous-time models to develop discrete-time non-linear models, again by using some approximation. This approach can be summarized as, first continuous-time linear models are developed for non-linear converters and later these continuous-time linear models are converted to discrete-time models. Considerable system behavior information is lost because of approximations used at both transitions.

A direct approach to problem should be employed. Techniques based on current-mode control are better suited for driving LEDs. Such techniques are based on current injection control that uses the fact that current injected into an inductor from input during the PWM on cycle will end up in output (with some pre-known scaling factor). This technique can increase the accuracy of the controlled parameter (current). Moreover, it can significantly reduce the complexity of the compensator. But these techniques also require a transformation from existing continuous-time models to discrete-time models.

A discrete-time non-linear control developed directly for these converters would certainly perform better. First, it can depict the exact behavior of system and none of the system behavior is lost in approximations. Secondly, one of the advan-

tages of digital power management is that it is very easy to implement non-linear control algorithms when compared with analog solution. Advantages of this capability can be fully reaped by using non-linear model to control power stages. Lastly these non-linear models are far less complex and a compensator for them can be realized using far less instructions. That, consequently, increases the upper limit on switching frequency. Switching frequency is a major limitation when compared to analog controllers. In short, use of such models will improve overall performance of digital power management solutions and enable them to really challenge analog counterparts. Unfortunately little work has been done in this regard.

The proposed current control model is based on same motivations. It is a non-linear discrete-time model proposed in the context of LEDs. There are two considerations in this regard. First, this uses the current mode control technique based current injection. Secondly, this model depicts non-linear behavior of converters. The behavior of the converter during the PWM on cycle is used to develop this model, while the PWM off cycle is not considered as it has no effect on the value of the output current. This modeling technique was only applied to the boost converter in our report. This model needs refinement and verification as it just proposed a way forward.

Some of the measurement techniques currently being used for analog control loop measurements cannot be directly employed for digital controllers. A new test setup is proposed to measure open-loop transfer functions for digital controllers. This test-setup performed really well and results are verifiable. But it can be further refined. The existing test setup is used for closed-loop method. Both open-loop and closed-loop measurements verified the models developed earlier with some limitation. Further work is suggested in improving test setups. Further work is suggested in transforming and interpreting well-known existing test setups for analog controllers to digital controllers.

This discussion can be concluded by a evaluation of TMS320F28035 MCU for such power control applications. This MCU is optimized for real-time control applications and offers two distinct features to support digital power control applications. First, a floating point coprocessor (CLA) is available for control-law implementation. Availability of two CPUs increases the performance considerably by increasing system bandwidth. Moreover, integration of powerful analog peripherals like ADC, PWM and comparator modules further enhances the capabilities. However, a number of limitations and improvements are discussed in 6.4. In short this controller is a complete package for digital power management solutions.

Bibliography

[1] *Headlamp without Optics Datasheet (LE UW D1W1 01 OSTAR)*, osram Semiconductor. [Online]. Available: http//:www.osram.com

[2] M. Day, "LED-driver considerations," 2004. [Online]. Available: http://www.ti.com/sc/analogapps

[3] *TMS320F28035 Data Sheet*. [Online]. Available: http//:www.ti.com

[4] E. Rogers, *Understanding Boost Power Stage in Switch Mode Power Stages (SLVA061)*, March 1999.

[5] *Using the Venable Windows Software Version 4 for Models 3215/3225/3235*. [Online]. Available: http://www.venable.biz

[6] "Using LEDs in Lighting," US Department of Energy, Tech. Rep. [Online]. Available: http://www1.eere.energy.gov/buildings/ssl/m/using_leds.html

[7] L. Balogh, "A Practical Introduction to Digital Power Control (SLUP232)," Texas Instruments, Tech. Rep., 2005.

[8] *TMS320F28035 Product Home Page*. [Online]. Available: http://focus.ti.com/docs/prod/folders/print/tms320f28035.html

[9] *How LEDs Work*. [Online]. Available: http://www1.eere.energy.gov/buildings/ssl

[10] S. Winder, *Power Supplies for LED Driving*. Elsevier Inc., 2008.

[11] "Haitzs law." [Online]. Available: http//:en.wikipedia.org/wiki/Haitz's_Law

[12] R. Lineback, "Solid State Lighting Set to Boost LED Growth," May 2006. [Online]. Available: www.ledsmagazine.com/features/3/5/6

[13] "Hella product page," light-Headlamps. [Online]. Available: www.hella.com

[14] "Visteon product specifications," light Emitting Diode (LED) Front Lighting.

[15] B. Flemming, "New Technologies in Electric-Powered Vehicles [Automotive Electronics]," *IEEE Vehicular Technology Magazine*, vol. 5, no. 1, March 2010.

[16] J. Betten, "Control Loop Consideration for an LED Driver," August 2007.

[17] B.-J. Huang, C.-W. Tang, and J.-H. Wu, "Study of System Dynamics of High-Power LEDs," in *International Conference on Electronic Materials and Packaging*, December 2006.

[18] D. Gacio, A. Calleja, J. Garcia, J. Ribas, and M. Rico-Secades, "Suitable Switching Converter Topologies for Automotive Signal Lamps and Headlamps Using Power LEDs," in *Industry Applications Society Annual Meeting, 2008. IAS '08. IEEE*, October 2008.

[19] E. Rogers, *Understanding Buck Power Stage in Switch Mode Power Stages (SLVA057)*, March 1999.

[20] R. Ridley, *Analyzing the SEPIC Converter*, November 2006. [Online]. Available: http//:www.ridleyengineering.com

[21] *C2000 real time microcontroller platform*. [Online]. Available: http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=1531&familyId=916

[22] *TMS320x2803x Piccolo Control Law Accelerator (CLA) Reference Guide-Rev. B(SPRUGE6)*. [Online]. Available: http://www-s.ti.com/sc/techlit/SPRUGE6

[23] *TMS320x2802x,2803 Piccolo Analog-to-Digital Converter and Comparator (SPRUGE5)*. [Online]. Available: http//www-s.ti.com/sc/techlit/SPRUGE5

[24] *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (SPRUGE9)*. [Online]. Available: http://www-s.ti.com/sc/techlit/SPRUGE9

[25] *TMS320x2802x, 2803 Piccolo High-Resolution Pulse-Width Modulator*. [Online]. Available: http://www-s.ti.com/sc/techlit/SPRUGE8

[26] V. Vorperian, "Simplified analysis of PWM converters using model of PWM switch. continuous conduction mode," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 26, no. 3, pp. 490 –496, May 1990.

[27] ——, "Simplified analysis of PWM converters using model of PWM switch. Discontinuous conduction mode," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 26, no. 3, pp. 497 –505, May 1990.

[28] Y.-T. Chang and Y.-S. Lai, "Effect of sampling frequency of A/D converter on controller stability and bandwidth of digital-controlled power converter," in *7th International Conference on Power Electronics*, October 2007, pp. 625 –629.

[29] M. Rashid, *Power Electronics Handbook*. Academic Press, 2006.

[30] S. Choudhury, *Designing a TMS320F280x Based Digitally Controlled DC-DC Switching Power Supply (SPRAAB3)*, July 2005. [Online]. Available: http://focus.ti.com/lit/an/spraab3/spraab3.pdf

[31] D. Sable and R. Ridley, "Comparison of performance of single-loop and current-injection control for PWM converters that operate in both continuous and discontinuous modes of operation," *IEEE Transactions on Power Electronics*, vol. 7, no. 1, pp. 136 –142, January 1992.

[32] Y.-S. Jung, J.-Y. Lee, and M.-J. Youn, "A new small signal modeling of average current mode control," in *29th Annual IEEE Power Electronics Specialists Conference*, vol. 2, May 1998, pp. 1118 –1124.

[33] *Control Suit*. [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/controlsuite.html

Appendix A

Matlab Models

A.1 Buck

```
%% Plant Parameters
Vin = 12.0;              %  Input Voltage
Vd = 3.3;                %  Led Forward voltage drop
N = 1;                   %  No of LEDs
Vo = N*Vd;               %  output Voltage
D= Vo/Vin;               %  Optimum Duty Cycle
Io  = 1;                 % Maximum Output Current
L = 10e-6;               % Inducter
C = 14.1e-6;             % Capitor
Rc = 0.031;              % Capcitor Parasitic Resistance
Rl = 0.047;              % Inducter Parasitic resistance
F = 500e3;               % Sampling Frequency
Ts = 1/F;                % Sampling Period
Td= 0.5*Ts;              % Time Delay from Sampling to Output Update
Kd= 1/3.3;
%% Factor Introduced due to Current Control
Rs = 0.02;               %  Shunt Resistance to measure current
Rd = 0.85;               %  Led Dynamic resistance
R = (Rd*N)+Rs;           %  Total Led Load Dynamic Resistance
Gamp = 30;               %  Current sense amplifier gain
Gi = Gamp*Rs/R;          % Transfer Function for Output Current to Output Voltage
Gdc = Vin*R/(R+Rl);      % Dc factor in Transfer function
%%%%%%%%%%%%%%% Numerator of Buck Converter Transfer Function %%%%%%%%
n1 = Rc*C;
n0 = 1;
num_Gps = Gdc*Gi*[n1 n0];
%%%%%%%%%%%%%%% Denominator of Buck Converter Transfer Function %%%%%%%%
d2 = (L*C*(R+Rc))/(R+Rl);
d1 = (C*(Rc+((R*Rl)/(R+Rl))))+(L/(R+Rl));
d0 =  1;
denom_Gps = [d2 d1 d0]; % Denominator
%% Continuos Tansfer Function
Gps_dly_Buck = tf(num_Gps,denom_Gps, inputdelay ,Td);
%% Continuos Tansfer Function
Gpz_Buck = c2d(Gps_dly_Buck*Kd,Ts, zoh );
sisotool( bode ,Gpz_Buck);
```

A.2 Boost

```
%% Plant Parameters
Vin = 12.1;       % Input Voltage
Vd = 2.906;       % Led Forward voltage drop
N = 10;           %  No of LEDs
Vo = N*Vd;        % output Voltage
```

```
D= 1-(Vin/Vo);    % Optimum Duty Cycle
Io = 1;           % Maximum Output Current
L = 39e-6;        % Inducter
Rl = 0.047;       % Inducter parasitic Resistance
C = 30e-6;        % Capacitor
Rc = 0.031;       % Parsitic Resistance
F = 500e3;        % Sampling Frequency
Ts = 1/F;         % Sampling Period
Td= 0.5*Ts;       % Delay from Sampling to output update
Kd = 1/3.3;
%% Factor Introduced due to Current Control
Rs = 0.1;           %  Shunt Resistance to measure current
Rd = 1.375;         %  Led Dynamic resistance
R =(Rd*N)+Rs;       %  Total Led Load Dynamic Resistance
Kamp = 20;          %  Amplifier gain
Gi = Rs*Kamp/R;     % Current factor
%% Plant Transfer Function
D1 = (1-D)^2;
Gd0 = Vin/D1;
Wz1 = 1/(Rc*C);
Wz2 = (D1*R-Rl)/L;
Wo = (1/sqrt(L*C))*sqrt((Rl+(D1*R))/R);
Q = Wo/((Rl/L)+(1/(C*(R+Rc))));
% Numerator of Boost Converter Transfer Function
n2 = -1/(Wz1*Wz2);
n1 = (1/Wz1)-(1/Wz2);
n0 = 1;
num_Gps = Gd0*Gi*[n2 n1 n0];
% Denominator of Boost1 Converter Transfer Function
d2 = 1/(Wo^2);
d1 = 1/(Wo*Q);
d0 = 1;
denom_Gps = [d2 d1 d0]; % Denominator
%% Continuos Tansfer Function
Gps_dly_Boost1 = tf(num_Gps,denom_Gps, inputdelay ,Td);
%% Continuos Tansfer Function
Gpz_Boost1 = c2d(Gps_dly_Boost1*Kd,Ts, zoh );
sisotool( bode ,Gpz_Boost1);
```

A.3 Buck-Boost

```
%% Plant Parameters %%%%%%%%%%%%%%%%%%%%%%%%%
Vin = 12.0;       % Input Voltage
Vd = 3.3;         % Led Forward voltage drop
N = 4;            %  No of LEDs
Vo = N*Vd;        % output Voltage
D = 1-(Vin/Vo);    % Optimum Duty Cycle
Io  = 1;          % Maximum Output Current
L = 39e-6;        % Inducter
Rl = 0.047;       % Inducter Parasitic Resistance
C = 30e-6;        % Capasitance
Rc = 0.031;       % Capacitor Parasitic Resistance
F = 300e3;        % Sampling Frequency
Ts = 1/F;         % Sampling Period
Td= 0.25*Ts;      % Delay from Sampling to Output Update
Kd=1/3.3;
%% Factor Introduced due to Current Control
Rs = 0.1;           %  Shunt Resistance to measure current
Rd = 0.85;          %  Led Dynamic resistance
R =(Rd*N)+Rs;       %  Total Led Load Dynamic Resistance
Kamp = 20;        % Amplifier gain
Gi = Rs*D*Kamp/R;   % current factor
%% Plant Transfer Fuction
D1 = (1-D)^2;
Gd0 = Vin/D1;
Wz1 = 1/(Rc*C);
Wz2 = (D1*(R-Rl))/L;
Wo = (1/sqrt(L*C))*sqrt((Rl+(D1*R))/R);
Q = Wo/((Rl/L)+(1/(C*(R+Rc))));
% Numerator of Buck-Boost Converter Transfer Function
n2 = -1/(Wz1*Wz2);
n1 = (1/Wz1)-(1/Wz2);
n0 = 1;
num_Gps = Gd0*Gi*[n2 n1 n0];
% Denominator of Buck-Boost Converter Transfer Function
d2 = 1/(Wo^2);
d1 = 1/(Wo*Q);
d0 = 1;
```

```
denom_Gps = [d2 d1 d0];          % Denominator
%% Continus Transfer Function
Gps_dly_Boost2 = tf(num_Gps,denom_Gps, inputdelay ,Td);
%% Discrete Transfer Function
Gpz_Boost2 = c2d(Gps_dly_Boost2*Kd,Ts, zoh );
sisotool( bode ,Gpz_Boost2);
```

A.4 SEPIC

```
%% Plant Parameters
Vin = 12.0;          % Input Voltage
Vd = 3.3;            % Led Forward voltage drop
N = 10;              % No of LEDs
Vo = N*Vd;           % output Voltage
D= Vo/(Vin+Vo);      % Optimum Duty Cycle
Io  = 1;             % Maximum Output Current
D1 = D^2;            % Duty Cycle Squre for simplifying
D2 = (1-D)^2;        % (1-Duty Cycle) Squre for simplifying
L1 = 15e-6;            % Inducter 1
L2 = 15e-6;            % Inducter 2
C1 = 14.1e-6;          % Capacitor 1
C2 = 30e-6;            % Capacitor 2
F = 300e3;             % Sampling Frequency
Ts = 1/F;              % Sampling Period
Td = 0.25*Ts;          % Delay from Sampling to Output Update
Kd= 1/3.3;
%% Factor Introduced due to Current Control
Rs = 0.1;              %  Shunt Resistance to measure current
Rd = 0.85;             %  Led Dynamic resistance
R = (Rd*N)+Rs;         %  Total Led Load Dynamic Resistance
Kamp = 20;             %  Amplifier gain
Gi = Rs*Kamp/R;
%% Plant Transfer Function
Gdc = 1/((1-D)^2);
Wz1 = (L1*D1)/(R*D2);
Wz2 = (D2*(L1+L2)*R*C1)/(L1*D1);
Wz3 = (L2*C1)/D;
A0 = (L1*C1*C2)/((L1*C1)+C2);
A1 = (L2*C1*C2)/((L2*C1*D2)+(C2*D1));
Wo1 = 1/sqrt((L1*((C2*D1/D2)+C1))+ (L2*(C1+C2)));
Wo2 = sqrt((1/A0)+(1/A1));
Q1 = (R*D2)/(Wo1*((L1*D1)+(L2*D2)));
Q2 = (R*C2*Wo2)/((L1+L2)*C1*(Wo1^2));
% Numerator of Boost1 Converter Transfer Function
n3 = -1*Wz1*Wz3;
n2 = (Wz1*Wz2) + Wz3;
n1 = -1*(Wz1+Wz2);
n0 = 1;
num_Gps = Gdc*Gi*[n3 n2 n1 n0];
% Denominator of Boost1 Converter Transfer Function
d4 = 1/((Wo1^2)*(Wo2^2));
d3 = (1/(Wo2*Q2*(Wo1^2)))+ (1/(Wo1*Q1*(Wo2^2)));
d2 = (1/(Wo1^2)) + (1/(Wo2^2)) + (1/(Wo1*Wo2*Q1*Q2));
d1 = (1/(Wo1*Q1))+ (1/(Wo2*Q2));
d0 = 1;
denom_Gps = [d4 d3 d2 d1 d0]; % Denominator
%% Continuos Tansfer Function
Gps_dly_Sepic = tf(num_Gps,denom_Gps, inputdelay ,Td);
%% Continuos Tansfer Function
Gpz_Sepic = c2d(Gps_dly_Sepic*Kd,Ts, zoh );
sisotool( bode ,Gpz_Sepic);
```

A.5 Filter Conversion Script

A.5.1 Convert.m

Script for converting compensator transfer function for inputting in 2nd order filter implemented in TMS320F28035 micro-controller.

```
Z0 = input( Enter Z0 );
Z1 = input( Enter Z1 );
Z2 = input( Enter Z2 );
P1 = input( Enter P1 );
P2 = input( Enter P2 );
x= 2^26;
B0 = Z0*x;
B1 = -Z0*(Z2+Z1)*x;
B2 = Z0*Z1*Z2*x;
A1 = (P1+P2)*x;
A2 = -1*P1*P2*x;
```

A.5.2 Convert2.m

Script for converting compensator transfer function for inputting in 3nd order filter implemented in TMS320F28035 micro-controller.

```
Z0 = input( Enter Z0 );
Z1 = input( Enter Z1 );
Z2 = input( Enter Z2 );
Z3 = input( Enter Z2 );
P1 = input( Enter P1 );
P2 = input( Enter P2 );
P3 = input( Enter P2 );
x= 2^26;
B0 = Z0*x;
B1 = -Z0*(Z3+Z2+Z1)*x;
B2 = x*Z0*((Z3*(Z1+Z2))+Z1*Z2);
B3 = -1*Z0*Z1*Z2*Z3*x;
A1 = (P1+P2+P3)*x;
A2 = -1*(P1*P2+ (P3*(P1+P2)))*x;
A3 = P1*P2*P3*x;
```

A.6 Current Mode Control

```
%% Plant Parameters
Vin = 12.0;       % Input Voltage
L = 39e-6;        % Inducter
Rl = 0.047;       % Inducter parasitic Resistance
F = 500e3;        % Sampling Frequency
Ts = 1/F;         % Sampling Period
Td= 0.5*Ts;       % Delay from Sampling to output update
%% Plant Transfer Function
Gdc = Vin*Ts/(2*Rl);
% Numerator of Boost Converter Transfer Function
n0 = 1;
num_Gps = Gdc*n0;
% Denominator of Boost1 Converter Transfer Function
d1 = L/Rl;
d0 = 1;
denom_Gps = [d1 d0]; % Denominator
%% Continuos Tansfer Function
Gps_dly_Boost1 = tf(num_Gps,denom_Gps, inputdelay ,Td);
%% Continuos Tansfer Function
Gpz_Boost1 = c2d(Gps_dly_Boost1,Ts, zoh );
sisotool( bode ,Gpz_Boost1);
```

Appendix B

Firmware

This section contains complete software implementation

B.1 Main.c

```
#include "LedDemo-Settings.h"
#include "PeripheralHeaderIncludes.h"
#include "DSP280x_EPWM_defines.h"
#if defined(_CLA_TYPE0)
    #include "LedDemo-CLAShared.h"
#endif

#include "DPLib.h"


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// FUNCTION PROTOTYPES
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// ------------------------------------ FRAMEWORK ---------------------------------------
void DeviceInit(void);
void ISR_Init(void);

#ifdef FLASH

void InitFlash();
void MemCopy(Uint16**, Uint16**, Uint16**);
#pragma CODE_SECTION(cla1_isr ,"ramfuncs");
#pragma CODE_SECTION(Pwm3_Dim ,"ramfuncs");
#pragma CODE_SECTION(Pwm6_Dim ,"ramfuncs");

#endif //(FLASH)

// State Machine function prototypes
//-----------------------------------
// Alpha states
void A0(void);   //state A0
void B0(void);   //state B0
void C0(void);   //state C0

// A branch states
void A1(void);   //state A1

// B branch states
void B1(void);   //state B1
void B2(void);   //state B2
void B3(void);   //state B3
void B4(void);   //state B4
void B5(void);   //state B5
```

```
// C branch states
void C1(void);   //state C1
void C2(void);   //state C2
void C3(void);   //state C3


// Variable declarations
void (*Alpha_State_Ptr)(void);  // Base States pointer
void (*A_Task_Ptr)(void);       // State pointer A branch
void (*B_Task_Ptr)(void);       // State pointer B branch
void (*C_Task_Ptr)(void);       // State pointer C branch


// ---------------------------------- USER ----------------------------------
void BuckSingle_CNF(int n, int prd, int mode, int phase);
void ADC_CascSeqCNF(int ChSel[], int TrigSel[], int ACQPS, int Conv, int mode);

void Dimming_CNF(int n, int period);

interrupt void cla1_isr(void);

void ISRC_init(void);

void trip1(void);
void trip2(void);
void trip3(void);
void trip4(void);

void ofset_meas(void);

/// dimming fuction/////////

void dimming(void);

void ref_check(void);

#if(Dimming == 2)
interrupt void Pwm3_Dim(void);
interrupt void Pwm6_Dim(void);
interrupt void Pwm7_Dim(void);
#endif

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

struct inputDS2P2Z {
    long  ofset;
    long  reference;
    float b2;
    float b1;
    float b0;
    float a2;
    float a1;
    float max;
    float min;
    };

struct inputDS3P3Z {
    long  ofset;
    long  reference;
    float b3;
    float b2;
    float b1;
    float b0;
    float a3;
    float a2;
    float a1;
    float max;
    float min;
    };

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// VARIABLE DECLARATIONS - GENERAL
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//--------------------------------Global Veriables--------------------------------//
int16 ofset1;    // Boost1 ofset
int16 ofset2;    // Boost2 ofset
int16 ofset4;    // Buck ofset
int16 ofset5;      // Sepic ofset

int16 Boost1_Threshold;              // Over Voltage Reference for Boost1
int16 Boost2_Threshold,Boost2Temp;       // Over Voltage REference for Boost2
```

```
int16 Buck_Threshold;              // Over Voltage Reference for Buck
int16 Sepic_Threshold;             // Over Voltage Reference for Sepic

Uint16 b1temp,b2temp,btemp,stemp;

// Global Flags for indicating OV Conditions Trigered in  CLA Task4
// ISR & Cleared in main loop with Clear OVP flags

int16 Boost1_OVF,Boost2_OVF,Buck_OVF,Sepic_OVF;

/////////////////user//////////

extern const int16 Boost1RefMax =  20480;  //  equal to 40 in Q9
extern const int16 Boost2RefMax  = 8704;   //  equal to 19 in Q9
extern const int16 BuckRefMax    = 15240;   //  equal to 7 in Q10
extern const int16 SepicRefMax   = 20480;  //  equal to 40 in Q9

//Scaling Constants (values found via spreadsheet; exact value calibrated per board)
int16   K_Vin,      K_Iin;
int16   K_Vsepic,   K_Isepic,   K_IsepicInd,    iK_Vsepic,     iK_Isepic;
int16   K_Vboost1,  K_Iboost1,  K_Iboost1Ind,   iK_Vboost1,    iK_Iboost1;
int16   K_Vboost2,  K_Iboost2,  K_Iboost2Ind,   iK_Vboost2,    iK_Iboost2;
int16   K_Vbuck,    iK_Vbuck,   K_Ibuck,        K_IbuckInd,    iK_Ibuck;

// ------------------------------------------ FRAMEWORK ------------------------------------------

int16   VTimer0[2];         // Virtual Timers slaved of CPU Timer 0 (A events)
int16   VTimer1[2];         // Virtual Timers slaved of CPU Timer 1 (B events)
int16   VTimer2[2];         // Virtual Timers slaved of CPU Timer 2 (C events)
int16   HRmode;

// ------------------------------------------ USER ------------------------------------------
// The CLA will work with floating point coefficients and will
// directly access the ADC result register

extern volatile Uint16 *CNTL_2P2Z_Fdbk1, *CNTL_2P2Z_Fdbk2, *CNTL_2P2Z_Fdbk4,*CNTL_3P3Z_CLA_Fdbk5;
 // for Sepic 3p-3z filter
extern struct inputDS2P2Z *CNTL_2P2Z_Coef1, *CNTL_2P2Z_Coef2, *CNTL_2P2Z_Coef4;
extern struct inputDS3P3Z *CNTL_3P3Z_CLA_Coef5;     // for Sepic 3p-3z filter

struct inputDS2P2Z Coef2P2Z_1, Coef2P2Z_2, Coef2P2Z_4;
struct inputDS3P3Z Coef2P2Z_5;

// The following are messages from the main CPU to the CLA

#pragma DATA_SECTION(Coef2P2Z_1, "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Coef2P2Z_2, "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Coef2P2Z_4, "CpuToCla1MsgRAM");
#pragma DATA_SECTION(Coef2P2Z_5, "CpuToCla1MsgRAM");

int16   SlewError, Duty[5];

int16   Vsepic,     VREFsepic,  DmaxSepic  ;
int16   Vboost1,    VREFboost1, DmaxBoost1  ;
int16   Vboost2,    VREFboost2, DmaxBoost2  ;
int16   Vbuck,      VREFbuck,   DmaxBuck  ;

int16   DmaxBoost1Dim,DmaxBoost2Dim,DmaxSepicDim;

float   DmaxBoost1Dimf,DmaxBoost2Dimf,DmaxSepicDimf;
float   DmaxBoost1f,DmaxBoost2f,DmaxSepicf;

long    UOUTsepic, UOUTboost1, UOUTboost2,  UOUTbuck  ;

int16   IREFsepic,  IREFboost1, IREFboost2, IREFbuck;
int16   IMAXsepic,  IMAXboost1, IMAXboost2, IMAXbuck;
int16   VMAXsepic,  VMAXboost1, VMAXboost2, VMAXbuck;

int16 IREFsepicTemp,    IREFboost1Temp, IREFboost2Temp, IREFbuckTemp;

int     ChSel[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int  TrigSel[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

// ASM Module Terminal pointers and variables

extern long *Buck_In1, *Buck_In2, *Buck_In4, *Buck_In5; // Buck_In3 not used (EPWM3)

short init_flag1,init_flag2,init_flag3,init_flag4;
short fin_flag1,fin_flag2,fin_flag3,fin_flag4;
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// VARIABLE DECLARATIONS - CCS WatchWindow / GUI support
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
// ------------------------------------ FRAMEWORK ---------------------------------------

//-----------------These variables will only be used while running from flash-----------------//

#ifdef FLASH

 int16   SerialCommsTimer,CommsOKflg;

 //Used for running BackGround in flash, and ISR in RAM

 extern Uint16 *RamfuncsLoadStart, *RamfuncsLoadEnd, *RamfuncsRunStart;
 extern Uint16 *Cla1ProgLoadStart, *Cla1ProgLoadEnd, *Cla1ProgRunStart;

//GUI support variables
// sets a limit on the amount of external GUI controls - increase as necessary

 int16 *varSetTxtList[16]; //16 textbox controlled variables
 int16 *varSetBtnList[16]; //16 button controlled variables
 int16 *varSetSldrList[16]; //16 slider controlled variables
 int16 *varGetList[16]; //16 variables sendable to GUI
 int16 *arrayGetList[16]; //16 arrays sendable to GUI

 Uint16 FbBuf[BUF_LEN];
 Uint16 PwmBuf[BUF_LEN];

 #pragma DATA_SECTION(FbBuf,"dataLog");
 #pragma DATA_SECTION(PwmBuf,"dataLog");

#endif

// ------------------------------------ USER ---------------------------------------

// Monitor ("Get")                          // Display as:
int16    Gui_Vin;                             // Q9
int16    Gui_Iin;                             // Q11
int16    Gui_Vsepic, Gui_Vboost1, Gui_Vboost2, Gui_Vbuck;     // Q9
int16    Gui_Isepic, Gui_Iboost1, Gui_Iboost2;            // Q14
int16    Gui_Ibuck;                           // Q11

//Set variables
int16    Gui_IsetSepic, Gui_IsetBoost1, Gui_IsetBoost2, Gui_IsetBuck;
int16    Gui_EnPWR;
int16    Gui_EnBlinker;

int16    PgainSepic, PgainBoost1, PgainBoost2, PgainBuck;         // "counts" (Q0)
int16    IgainSepic, IgainBoost1, IgainBoost2, IgainBuck;         // "counts" (Q0)
int16    DgainSepic, DgainBoost1, DgainBoost2, DgainBuck;         // "counts" (Q0)



// Variables for background support only (no need to access)
int16    i, HistPtr, temp_Scratch;       // Temp here means Temporary

// History arrays are used for Running Average calculation (boxcar filter)
// Used for CCS display and GUI only, not part of control loop processing
int16    VinH[HistorySize],IinH[HistorySize];
int16    VsepicH[HistorySize],Vboost1H[HistorySize];
int16  Voost2H[HistorySize],VbuckH[HistorySize];
int16    IsepicH[HistorySize],Iboost1H[HistorySize];
int16  Iboost2H[HistorySize],IbuckH[HistorySize];
int16    IsepicIndH[HistorySize],Iboost1IndH[HistorySize];
int16  Iboost2IndH[HistorySize],IbuckIndH[HistorySize];

int16  IinRCalib;

//-----------------------------------------------------------------------------------------//

int16  ClearBoost1_OVF;
int16  ClearBoost2_OVF;
int16  ClearBuck_OVF;
int16  ClearSepic_OVF;
int16  En_dim;
int16  temp2;
int16  Boost1_dim;
int16  Boost2_dim;
int16  Buck_dim;
int16  Sepic_dim;
int16  Boost1_dim_DC;
int16  Boost2_dim_DC;
int16  Buck_dim_DC;
int16  Sepic_dim_DC;
int16  pwm3_flag;
int16  pwm6_flag;
```

```
int16 s1,s2,s3,s4,s5,s6,s7,s8;

int16 Var;
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// MAIN CODE - starts here
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void main(void)
{
Var = 0;
//=================================================================================
//  INITIALISATION - General
//=================================================================================

//------------------------------------- FRAMEWORK -------------------------------------

DeviceInit();   // Device Life support & GPIO
// Only used if running from FLASH
#ifdef FLASH
// Copy time critical code and Flash setup code to RAM. The  RamfuncsLoadStart, RamfuncsLoadEnd
// and RamfuncsRunStart symbols are created by the linker. Refer to the linker files.
MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
// Call Flash Initialization to setup flash waitstates. This function must reside in RAM
InitFlash();    // Call the flash wrapper init function
#if defined(_CLA_TYPE0)
// Copy the CLA program code from its load address to the CLA program memory
MemCopy(&Cla1ProgLoadStart, &Cla1ProgLoadEnd, &Cla1ProgRunStart);
#endif //(_CLA_TYPE0)

//-------------------------------------
// Timing sync for background loops
// Timer period definitions found in PeripheralHeaderIncludes.h
CpuTimer0Regs.PRD.all =  mSec1;     // A tasks
CpuTimer1Regs.PRD.all =  mSec2;     // B tasks
CpuTimer2Regs.PRD.all =  mSec200;   // C tasks
// Tasks State-machine init
Alpha_State_Ptr = &A0;
A_Task_Ptr = &A1;
B_Task_Ptr = &B1;
C_Task_Ptr = &C1;
VTimer0[0] = 0;
VTimer1[0] = 0; VTimer1[1] = 0;
VTimer2[0] = 0;
HRmode = 1;     // Default to HR mode enabled
// ------------------------------------- USER -------------------------------------
 ClearBoost1_OVF=0;
 ClearBoost2_OVF=0;
 ClearBuck_OVF=0;
 ClearSepic_OVF=0;
//---------------------//
  Boost1_OVF=0;
  Boost2_OVF=0;
  Buck_OVF=0;
  Sepic_OVF=0;
 //---------------------//
  En_dim = 0;
  Boost1_dim = 0;
  Boost2_dim = 0;
  Buck_dim = 0;
  Sepic_dim = 0;
 //-------------------//
  init_flag1=1;
  init_flag2=1;
  init_flag3=1;
  init_flag4=1;
  fin_flag1=0;
  fin_flag2=0;
  fin_flag3=0;
  fin_flag4=0;
 //-------------------//
  Boost1_Threshold=20480;   //  equal to 40 in Q9
  Boost2_Threshold=8192;   //  equal to 19 in Q9
  Buck_Threshold=12288;     //  equal to 7 in Q10
  Sepic_Threshold=20480;    //  equal to 42 in Q9
  Boost1_dim_DC = 500;
  Boost2_dim_DC = 500;
  Buck_dim_DC = 500;
  Sepic_dim_DC = 500;
// For LED string board
DmaxSepic = 820;    DmaxBoost1 = 700;   DmaxBoost2 = 700;   DmaxBuck = 950;
DmaxSepicDim = 600; DmaxBoost1Dim = 600;    DmaxBoost2Dim = 600;
DmaxBoost1f = _IQ26toF(DmaxBoost1 * 67108);
DmaxBoost2f = _IQ26toF(DmaxBoost2 * 67108);
```

```
DmaxSepicf = _IQ26toF(DmaxSepic * 67108);
UOUTsepic = 0;UOUTboost1 = 0;UOUTboost2 = 0;UOUTbuck = 0;
IREFsepic = 0;IREFboost1 = 0;IREFboost2 = 0;IREFbuck = 0;
IREFsepicTemp = 0;IREFboost1Temp = 0;IREFboost2Temp = 0;IREFbuckTemp = 0;
// 2 pole / 2 Zero compensator coefficients (B2, B1, B0, A2, A1) are mapped to the simpler
// 3 coefficients P, I, D  to allow for trial & error intuitive tuning via CCS WatchWindow
// or GUI Sliders.  Note: User can modify if needed and assign full set of 5 coef.

PgainSepic = 1; IgainSepic = 1; DgainSepic = 5; // very "loose" initially
PgainBoost1 = 1;     IgainBoost1 = 1;     DgainBoost1 = 5;     // very "loose" initially
PgainBoost2 = 1;     IgainBoost2 = 1;     DgainBoost2 = 5;     // very "loose" initially
PgainBuck = 1;  IgainBuck = 1;  DgainBuck = 5;  // very "loose" initially

//-------------------------------------------------------------------------------------------------------/
//---------------------------------- Compensation implementation 1 ------------------------------------/
//-------------------------------------------------------------------------------------------------------/
#if (com == 1)

// Coefficient init for Boost1 Loop
Coef2P2Z_1.b2 = _IQ26toF(DgainBoost1 * 67108);
Coef2P2Z_1.b1 = _IQ26toF((IgainBoost1 - PgainBoost1 - DgainBoost1 - DgainBoost1)*67108);
Coef2P2Z_1.b0 = _IQ26toF((PgainBoost1 + IgainBoost1 + DgainBoost1)*67108);
Coef2P2Z_1.a2 = 0.0;
Coef2P2Z_1.a1 = 1.0;
Coef2P2Z_1.max = DmaxBoost1f;
Coef2P2Z_1.min = 0.0;
// Coefficient init for Boost2 Loop
Coef2P2Z_2.b2 = _IQ26toF(DgainBoost2 * 67108);
Coef2P2Z_2.b1 = _IQ26toF((IgainBoost2 - PgainBoost2 - DgainBoost2 - DgainBoost2)*67108);
Coef2P2Z_2.b0 = _IQ26toF((PgainBoost2 + IgainBoost2 + DgainBoost2)*67108);
Coef2P2Z_2.a2 = 0.0;
Coef2P2Z_2.a1 = 1.0;
Coef2P2Z_2.max = DmaxBoost2f;
Coef2P2Z_2.min= 0.0;

// Coefficient init for BUCK Loop
Cof2P2Z_4.b2 = _IQ26toF(DgainBuck * 67108);
Coef2P2Z_4.b1 = _IQ26toF((IgainBuck - PgainBuck - DgainBuck - DgainBuck)*67108);
Coef2P2Z_4.b0 = _IQ26toF((PgainBuck + IgainBuck + DgainBuck)*67108);
Coef2P2Z_4.a2 = 0.0;
Coef2P2Z_4.a1 = 1.0;
Coef2P2Z_4.max = _IQ26toF(DmaxBuck * 67108);
Coef2P2Z_4.min = 0.0;

// Coefficient init for SEPIC Loop

Coef2P2Z_5.b3 = 0.0;
Coef2P2Z_5.b2 = _IQ26toF(DgainSepic * 67108);
Coef2P2Z_5.b1 = _IQ26toF((IgainSepic - PgainSepic - DgainSepic - DgainSepic)*67108);
Coef2P2Z_5.b0 = _IQ26toF((PgainSepic + IgainSepic + DgainSepic)*67108);
Coef2P2Z_5.a3 = 0.0;
Coef2P2Z_5.a2 = 0.0;
Coef2P2Z_5.a1 = 1.0;
Coef2P2Z_5.max = DmaxSepicf;
Coef2P2Z_5.min= 0.0;
#endif

//-------------------------------------------------------------------------------------------------------/
HistPtr = 0;
//Configure Scaling Constants
K_Vin           = 18586;        // 0.567 in Q15 (see excel spreadsheet)
K_Iin           = 28160;        // Document this value! OKS
K_Vboost1       = 31861;        // 0.972 in Q15  (see excel spreadsheet)
K_Iboost1       = 27034;        // 0.825 in Q15  (see excel spreadsheet)
K_Iboost1Ind    = 22528;        // 0.688 in Q15  (see excel spreadsheet)
iK_Vboost1      = 16850;        // 1.028 in Q14 (see excel spreadsheet)
iK_Iboost1      = 19859;        // 1.212 in Q14 (see excel spreadsheet)
K_Vboost2       = 31861;        // 0.972 in Q15  (see excel spreadsheet)
K_Iboost2       = 27034;        // 0.825 in Q15  (see excel spreadsheet)
K_Iboost2Ind    = 22528;        // 0.688 in Q15  (see excel spreadsheet)
iK_Vboost2      = 16850;        // 1.028 in Q14 (see excel spreadsheet)
iK_Iboost2      = 19859;        // 1.212 in Q14 (see excel spreadsheet)
K_Vsepic        = 31861;        // 0.972 in Q15  (see excel spreadsheet)
K_Isepic        = 27034;        // 0.825 in Q15  (see excel spreadsheet)
K_IsepicInd     = 22528;        // 0.688 in Q15  (see excel spreadsheet)
iK_Vsepic       = 16850;        // 1.028 in Q14 (see excel spreadsheet)
iK_Isepic       = 19859;        // 1.212 in Q14 (see excel spreadsheet)
K_Vbuck         = 20275;            // 0.619 in Q15  (see excel spreadsheet)
K_Ibuck         = 22528;            // 0.688 in Q15  (see excel spreadsheet)
K_IbuckInd      = 29384;            // 0.897 in Q15  (see excel spreadsheet)
iK_Vbuck        = 26479;            // 1.616 in Q14 (see excel spreadsheet)
iK_Ibuck        = 23831;            // 1.455 in Q14 (see excel spreadsheet)
//=============================================================================
```

```
//   INITIALIZATION - Peripherals used for support
//===========================================================================
// ------------------------------------ USER ---------------------------------------------
// Configure the Control Law Accelerator
//--------------------------------------
#if defined(_CLA_TYPE0)
   //
   // This code assumes the CLA clock is already enabled in the call to InitSysCtrl();
   // The symbols used in this calculation are defined in the CLA assembly code and in
   // the CLAShared.h header file
   //
   EALLOW;
   //
   // Assign the CLA program memory to the CLA. This assumes it has already been
   // initialized by either the main CPU or by Code Composer Studio
   //
   Cla1Regs.MMEMCFG.bit.PROGE = 1;
   //
   // Initialize the CLA task vectors. Interrupt 1 will start task 1.
   Cla1Regs.MVECT1 = (Uint16) (&Cla1Task1 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MVECT2 = (Uint16) (&Cla1Task2 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MVECT3 = (Uint16) (&Cla1Task3 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MVECT4 = (Uint16) (&Cla1Task4 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MVECT7 = (Uint16) (&Cla1Task7 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MVECT8 = (Uint16) (&Cla1Task8 - &Cla1Prog_Start)*sizeof(Uint32);
   Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_ADCINT1;    // ADCINT1 will start CLA Task 1
   Cla1Regs.MPISRCSEL1.bit.PERINT2SEL = CLA_INT2_ADCINT2;    // ADCINT2 will start CLA Task 2
   Cla1Regs.MPISRCSEL1.bit.PERINT3SEL = CLA_INT3_ADCINT3;    // ADCINT3 will start CLA Task 3
   Cla1Regs.MPISRCSEL1.bit.PERINT4SEL = CLA_INT4_ADCINT4;    // ADCINT4 will start CLA Task 4
   EDIS;
#endif
//===========================================================================
//  INCREMENTAL BUILD OPTIONS - NOTE: select via ProjectSettings.h
//===========================================================================
// ------------------------------------ USER ---------------------------------------------
    IMAXsepic    = 16384;          //16384 = 1.0 in Q14
    IMAXboost1   = 16384;          //16384 = 1.0 in Q14
    IMAXboost2   = 16384;          //16384 = 1.0 in Q14
    IMAXbuck     = 28676;          //28676 = 0.7 in Q12
    Gui_EnPWR    = 0;
    Gui_EnBlinker = 0;
    Gui_IsetSepic   = 0; Gui_IsetBoost1 = 0;Gui_IsetBoost2 = 0;Gui_IsetBuck = 0;
    IinRCalib = 0;

// Have defined common ADC channel setup for all Incremental bilds

    #define      Iboost1R     AdcResult.ADCRESULT0     // ADC-B0   Boost1 FB
    #define      Iboost2R     AdcResult.ADCRESULT1     // ADC-B2   Boost2 FB
    #define      IbuckR   AdcResult.ADCRESULT2     // ADC-A3   Buck FB
      #define        IsepicR  AdcResult.ADCRESULT3     // ADC-B5   SEPIC FB
    #define      Vboost1R     AdcResult.ADCRESULT4     // ADC-B1   Boost1 output voltage
    #define      Vboost2R     AdcResult.ADCRESULT5     // ADC-B3   Boost2 output voltage
    #define      VbuckR   AdcResult.ADCRESULT6     // ADC-A5   Buck output voltage
    #define      VsepicR  AdcResult.ADCRESULT7     // ADC-B7   SEPIC output voltage
    #define      Iboost1IndR  AdcResult.ADCRESULT8     // ADC-A2   Boost1 switching current
    #define      Iboost2IndR  AdcResult.ADCRESULT9     // ADC-A4   Boost2 switching current
    #define      IbuckIndR    AdcResult.ADCRESULT10    // ADC-B4   Buck switching current
    #define      IsepicIndR   AdcResult.ADCRESULT11    // ADC-A6   SEPIC switching current
    #define      VinR         AdcResult.ADCRESULT12    // ADC-B6   Input voltage sense
    #define          IinR         AdcResult.ADCRESULT13    // ADC-A7   Input current monitor
// Configure the ADC
//--------------------------------------
    // Channel Selection for Cascaded Sequencer
    // This section is based on Board Layout so dont change it,
    // it will effect the portability of code to demo board

    ChSel[0] = 8;        // B0 - Boost 1 Feedback
    ChSel[1] = 10;       // B2 - Boost 2 Feedback
    ChSel[2] = 3;        // A3 - Buck Feedback
    ChSel[3] = 13;       // B5 - SEPIC Feedback
    ChSel[4] = 9;        // B1 - Boost1 output voltage
    ChSel[5] = 11;       // B3 - Boost2 output voltage
    ChSel[6] = 5;        // A5 - Buck output voltage
    ChSel[7] = 15;       // B7 - Sepic output voltage
    ChSel[8] = 2;        // A2 - Boost 1 switching current
    ChSel[9] = 4;        // A4 - Boost 2 switching current
    ChSel[10] = 12;      // B4 - Buck switching current
    ChSel[11] = 6;       // A6 - Sepic switching current
    ChSel[12] = 14;      // B6 - Input Voltage Sence
    ChSel[13] = 7;       // A7 - Input current sense
    ChSel[14] = 7;       // A7 - not used
    ChSel[15] = 6;       // A6 - not used
```

```
    TrigSel[0] = 5;          // B0 - Boost 1 Feedback
    TrigSel[1] = 7;          // B1 - Boost1  feedback
    TrigSel[2] = 11;         // B3 - Boost2 Output Voltage Sence Trigered by PWM2 SOC2A
    TrigSel[3] = 13;     // B7 - Sepic Output Voltage Sence Trigered by PWM5 SOC5A
    TrigSel[4] = 5;      // B0 - Boost 1 Feedback current Sence Trigered by PWM1 SOC1A
    TrigSel[5] = 7;      // B2 - Boost 2 Feedback Current Sence Trigered by PWM2 SOC2A
    TrigSel[6] = 11;     // B5 - Sepic Feedback Current Sence Trigered by PWM5 SOC5A
    TrigSel[7] = 13;         // B6 - Input Voltage Sence Trigered by PWM1 SOC1A
    TrigSel[8] = 5;      // B4 - Buck Switching Current Sence Trigered by PWM4 SOC4A
    TrigSel[9] = 7;      // A5 - Dummy Read Trigered by PWM1 SOC1A
    TrigSel[10] = 11;    // A3 - Buck Feedback Current Sence Trigered by PWM4 SOC4A
    TrigSel[11] = 13;    // A5 - Buck Output Voltage Sence Trigered by PWM4 SOC4A
    TrigSel[12] = 5;     // A2 - Boost 1 Switching Current Sence Trigered by PWM1 SOC1A
    TrigSel[13] = 5;     // A4 - Boost 2 Switching Current Sence Trigered by PWM2 SOC2A
    TrigSel[14] = 0;     // A7 - Input Current Sence Trigered by PWM1 SOC1A
    TrigSel[15] = 0;     // A6 - Sepic Switching Current Sence Trigered by PWM5 SOC5A
    ADC_CascSeqCNF(ChSel,TrigSel,16,10,0);
//-----------------------------------------------------------------------
#if (INCR_BUILD == 1)    // CLA Close Loop Boost1 V-Mode, no Dimming (INCR_BUILS 2 + CLA)
//-----------------------------------------------------------------------
    #define      prd1           90  // Period count = 333 kHz @ 60 MHz
BuckSingle_CNF(1, prd1, 1, 0);  // ePWM1 - Boost1,  Period=prd,  Master,  Phase= Don t Care
BuckSingle_CNF(2, prd1, 0, 90); // ePWM2 - Boost2,  Period=prd,  Slave,   Phase= 90 Degrees
BuckSingle_CNF(4, prd1, 0, 180);    // ePWM4 - Buck,    Period=prd,  Slave,   Phase= 180 Degrees
BuckSingle_CNF(5, prd1, 0, 270);    // ePWM5 - SEPIC,   Period=prd,  Slave,   Phase= 270 Degrees
//////////////////////////////////////////////////////////
#define  prd2    60000    // Period count = 3 kHz @ 60 MHz
#define phase    30000        // prd2*90/180
#if(Dimming == 2 )
Dimming_CNF(3,prd2);          // Configuration of Boost1 Dimming PWM
Dimming_CNF(6,prd2);         // Configuration of Sepic Dimming PWM
Dimming_CNF(7,prd2);         // Configuration of Boost2 Dimming PWM
 // Note:-
 // Boost2 Dimming FET is connected to EPWM-6B but we want to use PWM 7 for dimming
 // So we will put GPIO-11 on high impedence using trip zone
EPwm6Regs.TZCTL.bit.TZB = 0;    // enabling trip zone on PWM 6 output B
EPwm6Regs.TZCTL.bit.TZA = 3;    // enabling trip zone on PWM 6 output A
EPwm6Regs.TZFRC.bit.OST = 1;    // Triping Output B to put it in High Impedence

#endif

//////////////////////////////////////////////////////////////////////////

EALLOW;
AdcRegs.SOCPRICTL.bit.SOCPRIORITY = 0x004;  //SOCs 0-3 are high priority
EDIS;

    ISR_Init();                         // ASM ISR init

//////////////////////////////////////////////////////////////////////////


// CNTL_2P2Z connections
CNTL_2P2Z_Fdbk1 =&AdcResult.ADCRESULT0;             // point to Boost1 output voltage value
CNTL_2P2Z_Coef1 = &Coef2P2Z_1;              // point to first coeff of Boost1 Loop

CNTL_2P2Z_Fdbk2 = &AdcResult.ADCRESULT1;            // point to Boost2 output voltage value
CNTL_2P2Z_Coef2 = &Coef2P2Z_2;              // point to first coeff of Boost2 Loop

CNTL_2P2Z_Fdbk4 = &AdcResult.ADCRESULT2;            // point to Buck output voltage value
CNTL_2P2Z_Coef4 = &Coef2P2Z_4;              // point to first coeff of Buck Loop

CNTL_3P3Z_CLA_Fdbk5 = &AdcResult.ADCRESULT3;        // point to Sepic output voltage value
CNTL_3P3Z_CLA_Coef5 = &Coef2P2Z_5;             // point to first coeff of Sepic Loop

#endif // (INCR_BUILD == 1)



//=========================================================================
//  INTERRUPT & ISR INITIALISATION (best to run this section after other initialisation)
//=========================================================================
#if defined(_CLA_TYPE0)
    EALLOW;

// Enable CLA Task 8 and force it using the IACK instruction.
// This task will clear the DBUFF buffer used by the 2P2Z filter

Cla1Regs.MCTL.bit.IACKE = 1;
Cla1Regs.MIER.all =  M_INT8;
Cla1ForceTask8andWait();
asm("    RPT #3 || NOP");
```

```
// Enable CLA Task 1,2,3,4 and disable interrupt 8 CLA interrupt 1 is triggered by ADCINT1

Cla1Regs.MIER.bit.INT8 =  0;  // DISABLE TASK 8
Cla1Regs.MIER.bit.INT1 =  1;  // ENABLE TASK 1
Cla1Regs.MIER.bit.INT2 =  1;  // ENABLE TASK 2
Cla1Regs.MIER.bit.INT3 =  1;  // ENABLE TASK 3
Cla1Regs.MIER.bit.INT4 =  1;  // ENABLE TASK 4
asm("    RPT #3 || NOP");
EDIS;
#endif

EALLOW;
//////////// This interupt will triger CLA task1 for Boost 1 conversion ////////////
AdcRegs.INTSEL1N2.bit.INT1SEL = 0x00;           //interupt on EOC for boost 1 FB on B0 in SOC0
AdcRegs.INTSEL1N2.bit.INT1E = 0x1;          // enable Interrupt 1 in ADC
AdcRegs.INTSEL1N2.bit.INT1CONT = 0x1;

//////////// This interupt will triger CLA task2 for Boost 2 conversion ////////////
AdcRegs.INTSEL1N2.bit.INT2SEL = 0x01;           // triger interupt on EOC for boost 2 FB
AdcRegs.INTSEL1N2.bit.INT2E = 0x1;          // enable Interrupt 1 in ADC
AdcRegs.INTSEL1N2.bit.INT2CONT = 0x1;

//////////// This interupt will triger CLA task3 for Boost 2 conversion ////////////
AdcRegs.INTSEL3N4.bit.INT3SEL = 0x02;           // triger interupt on EOC for Sepic FB
AdcRegs.INTSEL3N4.bit.INT3E = 0x1;          // enable Interrupt 1 in ADC
AdcRegs.INTSEL3N4.bit.INT3CONT = 0x1;

//////////// This interupt will triger CLA task4 for Boost 2 conversion ////////////
AdcRegs.INTSEL3N4.bit.INT4SEL = 0x03;           // triger interupt on EOC for buck FB
AdcRegs.INTSEL3N4.bit.INT4E = 0x1;          // enable Interrupt 1 in ADC
AdcRegs.INTSEL3N4.bit.INT4CONT = 0x1;

//   ePWM interrupt just for checking sync
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO;       // INT on Zero event
EPwm1Regs.ETSEL.bit.INTEN = 0;            // Enable INT
EPwm1Regs.ETPS.bit.INTPRD = ET_1ST;         // Generate INT on every event

 #if(Dimming == 2)
  //      ePWM3 interrupt for Dimming
PieCtrlRegs.PIEIER3.bit.INTx3 = 1;
PieVectTable.EPWM3_INT = &Pwm3_Dim;
PieCtrlRegs.PIEIER3.bit.INTx6 = 1;
PieVectTable.EPWM6_INT = &Pwm6_Dim;
PieCtrlRegs.PIEIER3.bit.INTx7 = 1;
PieVectTable.EPWM7_INT = &Pwm7_Dim;
#endif
// Remap the Interrupt Vectors Being Used
PieVectTable.CLA1_INT4 = &cla1_isr;
PieCtrlRegs.PIEIER11.bit.INTx4 = 1;
EPwm1Regs.ETCLR.bit.INT=1;
EPwm3Regs.ETCLR.bit.INT=1;
AdcRegs.ADCINTFLGCLR.all = 0xFFFF;
PieCtrlRegs.PIEIFR11.all = 0x0000;
PieCtrlRegs.PIEACK.all = 0xFFFF;
PieCtrlRegs.PIECTRL.bit.ENPIE = 1;
IER |= M_INT11;                     // Enable CPU INT11 connected to CLA 1-8 INTs:
IER |= M_INT3;
asm(" CLRC INTM, DBGM");          // Enable Global interrupt INTM
EDIS;

//===========================================================================
//  BACKGROUND (BG) LOOP
//===========================================================================
//-------------------------------- FRAMEWORK --------------------------------
    for(;;)
    {
        // b1temp=AdcResult.ADCRESULT0 + boost1_ofset;
        // State machine entry & exit point
        //===================================================
        (*Alpha_State_Ptr)();    // jump to an Alpha state (A0,B0,...)
        //===================================================
    }
} //END MAIN CODE
//===========================================================================
//  STATE-MACHINE SEQUENCING AND SYNCRONIZATION
//===========================================================================

//-------------------------------- FRAMEWORK --------------------------------
void A0(void)
{
    // loop rate synchronizer for A-tasks
    if(CpuTimer0Regs.TCR.bit.TIF == 1)
```

```
    {
        CpuTimer0Regs.TCR.bit.TIF = 1;  // clear flag
        //-----------------------------------------------------------
        (*A_Task_Ptr)();          // jump to an A Task (A1,A2,A3,...)
        //-----------------------------------------------------------
        VTimer0[0]++;             // virtual timer 0, instance 0 (spare
        SerialCommsTimer++;
    }
    Alpha_State_Ptr = &B0;        // Comment out to allow only A tasks
}
void B0(void)
{
    // loop rate synchronizer for B-tasks
    if(CpuTimer1Regs.TCR.bit.TIF == 1)
    {
        CpuTimer1Regs.TCR.bit.TIF = 1;               // clear flag
        //-----------------------------------------------------------
        (*B_Task_Ptr)();          // jump to a B Task (B1,B2,B3,...)
        //-----------------------------------------------------------
        VTimer1[0]++;             // virtual timer 1, instance 0 (used to control SPI LEDs

    }
    Alpha_State_Ptr = &C0;
}
void C0(void)
{
    // loop rate synchronizer for C-tasks
    if(CpuTimer2Regs.TCR.bit.TIF == 1)
    {
        CpuTimer2Regs.TCR.bit.TIF = 1;               // clear flag
        //-----------------------------------------------------------
        (*C_Task_Ptr)();          // jump to a C Task (C1,C2,C3,...)
        //-----------------------------------------------------------
        VTimer2[0]++;             // virtual timer 2, instance 0 (spare)
    }

    Alpha_State_Ptr = &dimming;      // Back to State A0
}

void dimming(void)
{
if(VTimer0[0] == 50 )
    {
    VTimer0[0] = 0;
//-------------------------------PWM Based Dimming control-------------------------------//

#if(Dimming == 2)

if(En_dim == 1)
{
EALLOW;
//-------------------------------------------------------------------------------------------------------//
//---------- Routine for enabling/Disabling Boost1 and Buck Dimming on PWM Channel 3 ----------//

if(Boost1_dim == 1 )
{
temp2=(prd2/1000)*Boost1_dim_DC;
EPwm3Regs.CMPA.half.CMPA = temp2;
EPwm3Regs.CMPB = temp2-5;
ifinit_flag1 == 1)
             {
             GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 1;
             EPwm3Regs.ETSEL.bit.INTEN = 1;
             fin_flag1=1;
             init_flag1 = 0;
                        }
}
else
{
if(fin_flag1==1)
{
             EPwm3Regs.ETSEL.bit.INTEN = 0;
             fin_flag1 = 0;
             init_flag1 = 1;
             GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 0;              // 0=GPIO,  1=EPWM3A,  2=Resv,  3=Resv
             GpioCtrlRegs.GPADIR.bit.GPIO4 = 1;               // 1=OUTput,  0=INput
             GpioDataRegs.GPASET.bit.GPIO4 = 1;               // uncomment if --> Set High initially
             EPwm1Regs.TZCLR.bit.OST = 1;
}
    }
//-------------------------------------------------- END ----------------------------------------------------//
//-------------------------------------------------------------------------------------------------------//
//---------------- Routine for enabling/Disabling Sepic Dimming on PWM Channel 6 ----------------//
```

```
if(Boost2_dim == 1 )
{
temp2=(prd2/1000)*Boost2_dim_DC;
EPwm7Regs.CMPA.half.CMPA = temp2;
EPwm7Regs.CMPB = temp2-5;
if(init_flag2 == 1)
{
EPwm7Regs.ETSEL.bit.INTEN = 1;
GpioCtrlRegs.GPBMUX1.bit.GPIO40 = 1;
fin_flag2 = 1;
init_flag2 = 0;
}
}
else
{
if(fin_flag2==1)
{
EPwm7Regs.ETSEL.bit.INTEN = 0;
fin_flag2=0;
init_flag2 = 1;
Boost2_dim = 0;
GpioCtrlRegs.GPBMUX1.bit.GPIO40 = 0;
GpioCtrlRegs.GPBDIR.bit.GPIO40 = 1;
GpioDataRegs.GPBSET.bit.GPIO40 = 1;
EPwm2Regs.TZCLR.bit.OST = 1;
}
}
//-------------------------------------------------- END ----------------------------------------------------//
//-----------------------------------------------------------------------------------------------------------//
//---------------- Routine for enabling/Disabling Sepic Dimming on PWM Channel 6 ----------------//
if(Sepic_dim == 1 )
{
temp2=(prd2/1000)*Sepic_dim_DC;
EPwm6Regs.CMPA.half.CMPA = temp2;
EPwm6Regs.CMPB = temp2-5;
if(init_flag4 == 1)
{
EPwm6Regs.ETSEL.bit.INTEN = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO10 = 1;
fin_flag4 = 1;
init_flag4 = 0;
}
}
else
{
if(fin_flag4==1)
{
EPwm6Regs.ETSEL.bit.INTEN = 0;
fin_flag4 = 0;
init_flag4 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO10 = 0;          // 0=GPIO,  1=EPWM3A,  2=Resv,  3=Resv
GpioCtrlRegs.GPADIR.bit.GPIO10 = 1;           // 1=OUTput,  0=INput
GpioDataRegs.GPASET.bit.GPIO10 = 1;           // uncomment if --> Set High initially
EPwm5Regs.TZCLR.bit.OST = 1;
}
}
//-------------------------------------------------END-----------------------------------------------------//
EDIS;
DmaxBoost1Dimf = _IQ26toF(DmaxBoost1Dim * 67108);
DmaxBoost2Dimf = _IQ26toF(DmaxBoost2Dim * 67108);
DmaxSepicDimf = _IQ26toF(DmaxSepicDim * 67108);
}
#endif
//---------------------------- END PWM Based Dimming control ----------------------------//
} // end Vtimer loop
Alpha_State_Ptr = &A0;
}
//===========================================================================
//  A - TASKS
//===========================================================================
//-----------------------------------------------------
void A1(void) // SMPS power On/Off Control, DMAX Clamping, Soft start
//-----------------------------------------------------
// Task runs every 1ms (CpuTimer0 period [1ms] * 1 "A" tasks active)
{
    if (Gui_EnPWR == 1)
       {
        GpioDataRegs.GPBSET.bit.GPIO44 = 1;      // Enable VPWR
            ref_check();
       }
    else
    {
            asm("NOP");
```

```
        GpioDataRegs.GPBCLEAR.bit.GPIO44 = 1;    // Disable VPWR
        IinRCalib = IinR;
    }

    //-----------------------
    A_Task_Ptr = &A2;
    //-----------------------
}
//---------------------------------------------------------------
void A1(void) // SMPS power On/Off Control, DMAX Clamping, Soft start
//---------------------------------------------------------------
// Task runs every 1ms (CpuTimer0 period [1ms] * 1 "A" tasks active)
{
//---------------------------------------------------------------
        if(Boost1_OVF == 1)
        {
        IREFboost1 = 0;
        Gui_IsetBoost1 =0;
        Boost1_dim = 0;
        }
//---------------------------------------------------------------
        if(Boost2_OVF == 1)
        {
        IREFboost2 = 0;
        Gui_IsetBoost2 =0;
        Boost2_dim = 0;
        }
//---------------------------------------------------------------
        if( Buck_OVF == 1)
        {
        IREFbuck = 0;
        Gui_IsetBuck =0;
    //  IsetBuckTemp=0;
        Buck_dim = 0;
        }
//---------------------------------------------------------------
        if(Sepic_OVF == 1)
        {
        IREFsepic = 0;
        Gui_IsetSepic =0;
        Sepic_dim = 0;
        }
    //-----------------------
    A_Task_Ptr = &A1;
    //-----------------------
}

//===========================================================================
//  B - TASKS
//===========================================================================
//---------------------------------- USER -----------------------------------
//---------------------------------------------------------------
void B1(void) // Input Voltage & Current  Dashboard measurements
//---------------------------------------------------------------
{
// Task runs every 10ms (CpuTimer1 period [5ms] * 5 "B" tasks active)
//
// Voltage measurement calculated by:
//  Gui_Vin = VinAvg * K_Vin, where VinAvg = sum of 8 VinR samples
//
    HistPtr++;
    if (HistPtr >= HistorySize)  HistPtr = 0;

    // BoxCar Averages - Input Raw samples into BoxCar arrays
    //-----------------------------------------------------------------

    VinH[HistPtr]           = VinR;
    IinH[HistPtr]           = IinR - IinRCalib;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + VinH[i];
    Gui_Vin = ( (long) temp_Scratch * (long) K_Vin ) >> 15;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + IinH[i];

    Gui_Iin = ( (long) temp_Scratch * (long) K_Iin ) >> 15;



    //-------------------
    B_Task_Ptr = &B2;
    //-------------------
```

```
}


//---------------------------------------------------------------
void B2(void) //  Sepic  Dashboard measurements
//---------------------------------------------------------------
// Task runs every 10ms (CpuTimer1 period [2ms] * 5 "B" tasks active)
{
    //- Checks and set the cuurent if its greater the Max Allowable

    if (Gui_IsetSepic > IMAXsepic)  Gui_IsetSepic = IMAXsepic;

// Assign the new value of Gui_IsetSepic to IREFSepic

    Coef2P2Z_5.reference =  ((long) Gui_IsetSepic * (long) iK_Isepic )>> 14;

    if(Gui_EnPWR == 1)       IsepicH[HistPtr]   = IsepicR + Coef2P2Z_5.ofset ;
    else                     IsepicH[HistPtr]   = IsepicR ;

    VsepicH[HistPtr]         = VsepicR;
    IsepicIndH[HistPtr]      = IsepicIndR;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + VsepicH[i];
    Gui_Vsepic = ( (long) temp_Scratch * (long) K_Vsepic ) >> 15;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + IsepicH[i];
    if(Gui_EnPWR == 0)  Coef2P2Z_5.ofset = temp_Scratch >> 3;
    Gui_Isepic = ( (long) temp_Scratch * (long) K_Isepic ) >> 15;

    //--------------------
    B_Task_Ptr = &B3;
    //--------------------
}

//---------------------------------------------------------------
void B3(void) //  Boost1  Dashboard measurements
//---------------------------------------------------------------
// Task runs every 10ms (CpuTimer1 period [2ms] * 5 "B" tasks active)
{

    //- Checks and set the cuurent if its greater the Max Allowable

    if (Gui_IsetBoost1 > IMAXboost1)

    Gui_IsetBoost1 = IMAXboost1;

// Assign the new value of Giu_IsetBoost1 to IREFboost1

     Coef2P2Z_1.reference =  ( (long) Gui_IsetBoost1 * (long) iK_Iboost1 )>> 14;

 if(Gui_EnPWR == 1)          Iboost1H[HistPtr]   = Iboost1R + Coef2P2Z_1.ofset ;
else                     Iboost1H[HistPtr]   = Iboost1R ;

 Vboost1H[HistPtr]       = Vboost1R;

temp_Scratch=0;
for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + Vboost1H[i];
Gui_Vboost1 = ( (long) temp_Scratch * (long) K_Vboost1 ) >> 15;

temp_Scratch=0;
for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + Iboost1H[i];
if(Gui_EnPWR == 0)  Coef2P2Z_1.ofset = temp_Scratch >> 3;
Gui_Iboost1 = ( (long) temp_Scratch * (long) K_Iboost1 ) >> 15;




    //--------------------
    B_Task_Ptr = &B4;
    //--------------------
}

//---------------------------------------------------------------
void B4(void) //  Boost2  Dashboard measurements
//---------------------------------------------------------------
// Task runs every 10ms (CpuTimer1 period [2ms] * 5 "B" tasks active)
{
//- Checks and set the cuurent if its greater the Max Allowable

if (Gui_IsetBoost2 > IMAXboost2)     Gui_IsetBoost2 = IMAXboost2;
```

```
// Assign the new value of Gui_IsetBoost2 to IREFboost2

Coef2P2Z_2.reference = ((long) Gui_IsetBoost2 * (long) iK_Iboost2 ) >> 14;

if(Gui_EnPWR == 1)       Iboost2H[HistPtr]  = Iboost2R + Coef2P2Z_2.ofset  ;
else                     Iboost2H[HistPtr]  = Iboost2R ;

Vboost2H[HistPtr]        = Vboost2R ;

Iboost2IndH[HistPtr]     = Iboost2IndR;

temp_Scratch=0;
for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + Vboost2H[i];
Gui_Vboost2 = ( (long) temp_Scratch * (long) K_Vboost2 ) >> 15;

if(Gui_EnPWR == 1) Gui_Vboost2 = Gui_Vboost2 - Gui_Vin; // For daetails check schemtic

temp_Scratch=0;
for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + Iboost2H[i];
if(Gui_EnPWR == 0)  Coef2P2Z_2.ofset = temp_Scratch >> 3;
Gui_Iboost2 = ( (long) temp_Scratch * (long) K_Iboost2 ) >> 15;



    //-------------------
    B_Task_Ptr = &B5;
    //-------------------
}

//---------------------------------------------------------------
void B5(void) //  Buck Dashboard measurements
//---------------------------------------------------------------
// Task runs every 10ms (CpuTimer1 period [2ms] * 5 "B" tasks active)
{

//- Checks and set the cuurent if its greater the Max Allowable

    if (Gui_IsetBuck > IMAXbuck)     Gui_IsetBuck = IMAXbuck;

// Assign the new value of Gui_IsetBuck to IsetBuckTemp

    Coef2P2Z_4.reference = ( (long) Gui_IsetBuck * (long) iK_Ibuck ) >> 14;

    VbuckH[HistPtr]   = VbuckR;

    if(Gui_EnPWR == 1)       IbuckH[HistPtr]   = IbuckR + Coef2P2Z_4.ofset ;
    else                     IbuckH[HistPtr]   = IbuckR ;

    IbuckIndH[HistPtr]       = IbuckIndR;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + VbuckH[i];
    Gui_Vbuck = ( (long) temp_Scratch * (long) K_Vbuck ) >> 15;

    temp_Scratch=0;
    for(i=0; i<8; i++)  temp_Scratch = temp_Scratch + IbuckH[i];
    if(Gui_EnPWR == 0)  Coef2P2Z_4.ofset = temp_Scratch >> 3;
    Gui_Ibuck = ( (long) temp_Scratch * (long) K_Ibuck ) >> 15;

    //-------------------
    B_Task_Ptr = &B1;
    //-------------------
}


//===========================================================================
//  C - TASKS
//===========================================================================

//---------------------------------- USER -----------------------------------------------
//---------------------------------------------------------------



//-----------------------------------------
void C1(void) //  Blinker control (Buck)
//-----------------------------------------
// Task runs every 1000ms (CpuTimer2 period [500ms] * 2 "C" tasks active)
{

    // toggle an LED on the controlCARD
```

```
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;   // 1/(2*500ms) = 1Hz blink rate

    //-------------------
    C_Task_Ptr = &C2;
    //-------------------
}

//-----------------------------------------
void C2(void) //  Trip zone clear task
//-----------------------------------------
// Task runs every 1000ms (CpuTimer2 period [500ms] * 2 "C" tasks active)
{

    if (ClearBoost1_OVF==1)
    {
        EALLOW;
        EPwm1Regs.TZCLR.bit.OST = 1;
        EDIS;
        ClearBoost1_OVF=0;
        Boost1_OVF =0;

    }
    if (ClearBoost2_OVF==1)
    {
        EALLOW;
        EPwm2Regs.TZCLR.bit.OST = 1;
        EDIS;
        ClearBoost2_OVF=0;
        Boost2_OVF =0;

    }

    if (ClearBuck_OVF==1)
    {
        EALLOW;
        EPwm4Regs.TZCLR.bit.OST = 1;
        EDIS;
        ClearBuck_OVF=0;
        Buck_OVF =0;

    }

    if (ClearSepic_OVF==1)
    {
        EALLOW;
        EPwm5Regs.TZCLR.bit.OST = 1;
        EDIS;
        ClearSepic_OVF=0;
        Sepic_OVF =0;
    }
        //-------------------
C_Task_Ptr = &C1;
//-------------------
}


void ofset_meas(void)
{
ofset1=AdcResult.ADCRESULT0 ;           //  For Boost1 FeedBack
ofset2=AdcResult.ADCRESULT1 ;           //  For Boost2 FeedBack
ofset4=AdcResult.ADCRESULT2;        //  For Buck FeedBack
ofset5=AdcResult.ADCRESULT3 ;           //  For Sepic FeedBack
}

void ref_check()
{
if(Boost1_Threshold > Boost1RefMax)         Boost1_Threshold = Boost1RefMax ;
if(Boost2_Threshold > Boost2RefMax)         Boost2_Threshold = Boost2RefMax ;
temp2= ((long)AdcResult.ADCRESULT12 * (long) K_Vin ) >> 12;;
Boost2Temp = Boost2_Threshold + temp2;
if(Buck_Threshold > BuckRefMax)             Buck_Threshold = BuckRefMax ;
if(Sepic_Threshold > SepicRefMax)           Sepic_Threshold = SepicRefMax ;
}

#if(Dimming == 2)

void interrupt Pwm3_Dim(void)
{
EALLOW;
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;

if(EPwm3Regs.ETSEL.bit.INTSEL == ET_CTRD_CMPA)
{
```

```
EPwm1Regs.TZCLR.bit.OST = 1;
EPwm3Regs.ETSEL.bit.INTSEL = ET_CTRU_CMPB;
EPwm1Regs.ETSEL.bit.SOCAEN =1;
}
else if(EPwm3Regs.ETSEL.bit.INTSEL == ET_CTRU_CMPB)
{
EPwm1Regs.TZFRC.bit.OST = 1;
EPwm3Regs.ETSEL.bit.INTSEL = ET_CTRD_CMPA;
EPwm1Regs.ETSEL.bit.SOCAEN =0;
}
EPwm3Regs.ETCLR.bit.INT=1;
EDIS;
}


void interrupt Pwm6_Dim(void)
{
EALLOW;
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;

if(EPwm6Regs.ETSEL.bit.INTSEL == ET_CTRD_CMPA)
{
EPwm5Regs.TZCLR.bit.OST = 1;
EPwm6Regs.ETSEL.bit.INTSEL = ET_CTRU_CMPB;
EPwm5Regs.ETSEL.bit.SOCAEN =1;
}
else if(EPwm6Regs.ETSEL.bit.INTSEL == ET_CTRU_CMPB)
{
EPwm5Regs.TZFRC.bit.OST = 1;
EPwm6Regs.ETSEL.bit.INTSEL = ET_CTRD_CMPA;
EPwm5Regs.ETSEL.bit.SOCAEN =0;
}
EPwm6Regs.ETCLR.bit.INT=1;
EDIS;
}



void interrupt Pwm7_Dim(void)
{
EALLOW;
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
if(EPwm7Regs.ETSEL.bit.INTSEL == ET_CTRD_CMPA)
{
EPwm2Regs.TZCLR.bit.OST = 1;
EPwm7Regs.ETSEL.bit.INTSEL = ET_CTRU_CMPB;
EPwm5Regs.ETSEL.bit.SOCAEN =1;
}
else if(EPwm7Regs.ETSEL.bit.INTSEL == ET_CTRU_CMPB)
{
EPwm2Regs.TZFRC.bit.OST = 1;
EPwm7Regs.ETSEL.bit.INTSEL = ET_CTRD_CMPA;
EPwm5Regs.ETSEL.bit.SOCAEN =0;
}
EPwm7Regs.ETCLR.bit.INT=1;
EDIS;
}

#endif
```

B.2 2nd Order IIR Filter code

```
;=====================================
IIR2P2Z_INIT     .macro n
;=====================================
; Variable Declarations
; The following are messages from the main CPU to the CLA
_CNTL_2P2Z_Fdbk:n:   .usect "CpuToCla1MsgRAM",2
_CNTL_2P2Z_Coef:n:   .usect "CpuToCla1MsgRAM",2
; variables for the CLA using CLA-to-CPU Ram as Data Ram for CLA
_CNTL_2P2Z_DBUFF:n:       .usect "Cla1ToCpuMsgRAM",10
;Publish Terminal Pointers for access from the C environment
.def     _CNTL_2P2Z_Fdbk:n:
.def     _CNTL_2P2Z_Coef:n:
.def     _CNTL_2P2Z_DBUFF:n:
; set terminal to point to ZeroNet
MOVL     XAR2, #ZeroNetCLA
MOVW     DP, #_CNTL_2P2Z_Coef:n:
```

```
MOVL    @_CNTL_2P2Z_Fdbk:n:, XAR2
MOVL    @_CNTL_2P2Z_Coef:n:, XAR2
.endm

;=====================================
IIR2P2Z     .macro n
;=====================================
_ControlLaw_2P2Z_Start:n:

        MMOV16  MAR0, @_CNTL_2P2Z_Coef:n:
        MMOV16  MAR1, @_CNTL_2P2Z_Fdbk:n:
        MNOP    ; can not use MAR0
        MNOP    ; can not use MAR1
        MUI32TOF32 MR3, *MAR0[#2]++
        MUI16TOF32 MR1, *MAR1
        MADDF32    MR1,MR1,MR3
        MMPYF32    MR1, MR1, #(1.0L/4096.0L)
        MUI32TOF32 MR0, *MAR0[#2]++
        MMPYF32    MR0, MR0, #(1.0L/32768.0L)
      MSUBF32   MR0, MR0, MR1
        MMOV32     @_CNTL_2P2Z_DBUFF:n:+4, MR0
        MMOV32     MR0, @_CNTL_2P2Z_DBUFF:n:+8
        MMOV32     MR1, *MAR0[#2]++
        MMPYF32    MR3, MR0, MR1
      || MMOV32    MR1, *MAR0[#2]++
        MMOVD32    MR0, @_CNTL_2P2Z_DBUFF:n:+6
        MMPYF32    MR2, MR0, MR1
        MMOV32     MR1, *MAR0[#2]++
        MMOVD32    MR0, @_CNTL_2P2Z_DBUFF:n:+4
        MMPYF32    MR2, MR1, MR0
      || MADDF32   MR3, MR3, MR2
        MMOV32     MR0, @_CNTL_2P2Z_DBUFF:n:+2
        MMOV32     MR1, *MAR0[#2]++
        MMPYF32    MR2, MR0, MR1
      || MADDF32   MR3, MR3, MR2
        MMOVD32    MR0, @_CNTL_2P2Z_DBUFF:n:+0
        MMOV32     MR1, *MAR0[#2]++
        MMPYF32    MR2, MR0, MR1
      || MADDF32   MR3, MR3, MR2
        MADDF32    MR3, MR3, MR2
          MUI16TOF32 MR0, @_EPwm:n:Regs.TBPRD
          MMINF32    MR3, MR1
        MMAXF32    MR3, #0.0
        MMOV32     @_CNTL_2P2Z_DBUFF:n:+0, MR3
;/////////PWM Update Section  ////////////////
        MMPYF32     MR0, MR0, MR3
        MMPYF32   MR1,MR0,#65536.0L
      MF32TOI32   MR1, MR1
      MMOV32      @_EPwm:n:Regs.CMPA.all, MR1

_ControlLaw_2P2Z_End:n:
 .endm

;=====================================
IIR2P2Z_DBUFF_CLA_INIT  .macro n
;=====================================
; Use this macro in a task forced using software
; to perform the init.
.if
```

B.3 3rd Order IIR Filter code

```
;=====================================
IIR3P3Z_INIT    .macro n
;=====================================

_CNTL_3P3Z_CLA_Fdbk:n:      .usect "CpuToCla1MsgRAM",2,1,1
_CNTL_3P3Z_CLA_Coef:n:      .usect "CpuToCla1MsgRAM",2,1,1
_CNTL_3P3Z_CLA_DBUFF:n:     .usect "Cla1ToCpuMsgRAM",14,1,1
;Publish Terminal Pointers for access from the C environment
.def    _CNTL_3P3Z_CLA_Fdbk:n:
.def    _CNTL_3P3Z_CLA_Coef:n:
.def    _CNTL_3P3Z_CLA_DBUFF:n:
; set terminal to point to ZeroNet
MOVL    XAR2, #ZeroNetCLA
MOVW    DP, #_CNTL_3P3Z_CLA_Coef:n:
MOVL    @_CNTL_3P3Z_CLA_Fdbk:n:, XAR2
MOVL    @_CNTL_3P3Z_CLA_Coef:n:, XAR2
```

```
.endm

;====================================
IIR3P3Z       .macro n
;====================================
_ControlLaw_3P3Z_Start:n:

MMOV16  MAR0, @_CNTL_3P3Z_CLA_Coef:n:
MMOV16  MAR1, @_CNTL_3P3Z_CLA_Fdbk:n:
MNOP  ; can not use MAR0
MNOP  ; can not use MAR1
MUI32TOF32 MR3, *MAR0[#2]++
MUI16TOF32 MR1, *MAR1
MADDF32     MR1,MR1,MR3
MMPYF32     MR1, MR1, #(1.0L/4096.0L)
MUI32TOF32 MR0, *MAR0[#2]++
MMPYF32     MR0, MR0, #(1.0L/32768.0L)
MSUBF32     MR0, MR0, MR1
MMOV32      @_CNTL_3P3Z_CLA_DBUFF:n:+6, MR0
MMOV32      MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+12
MMOV32      MR1, *MAR0[#2]++
MMPYF32     MR3, MR0, MR1
|| MMOV32      MR1, *MAR0[#2]++
MMOVD32     MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+10
MMPYF32     MR2, MR0, MR1
|| MMOV32      MR1, *MAR0[#2]++
MMOVD32     MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+8
MMPYF32     MR2, MR1, MR0
|| MADDF32     MR3, MR3, MR2
MMOVD32      MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+6
MMOV32      MR1, *MAR0[#2]++
MMPYF32     MR2, MR0, MR1
|| MADDF32     MR3, MR3, MR2
MMOV32     MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+4
MMOV32      MR1, *MAR0[#2]++
MMPYF32     MR2, MR0, MR1
|| MADDF32     MR3, MR3, MR2
MMOVD32     MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+2
MMOV32      MR1, *MAR0[#2]++
MMPYF32     MR2, MR0, MR1
|| MADDF32     MR3, MR3, MR2
MMOVD32     MR0, @_CNTL_3P3Z_CLA_DBUFF:n:+1
MMOV32      MR1, *MAR0[#2]++
MMPYF32     MR2, MR0, MR1
|| MADDF32     MR3, MR3, MR2
MADDF32     MR3, MR3, MR2                   ; 1
MMOV32      MR1, *MAR0[#2]++
MUI16TOF32   MR0,   @_EPwm:n:Regs.TBPRD
MMINF32     MR3,   MR1
MMAXF32     MR3,   #0.0
MMOV32      *MAR1[#0]++, MR3
MMOV32      @_CNTL_3P3Z_CLA_DBUFF:n:+0, MR3
MMPYF32      MR0, MR0, MR3
MMPYF32      MR1,MR0,#65536.0L
MF32TOI32    MR1, MR1
MMOV32       @_EPwm:n:Regs.CMPA.all, MR1

_ControlLaw_3P3Z_End:n:
 .endm
;-----------------------------------------------------------------------------------------------
;====================================
IIR3P3Z_DBUFF_CLA_INIT   .macro n
;====================================
; Initialize the CLA writable DATA buffer
_ControlLaw_3P3Z_DBUFF_CLA_INIT_START:n:

MSETFLG       RNDF32=1
MMOVIZ        MR0, #0.0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+0, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+2, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+4, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+6, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+8, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+10, MR0
MMOV32        @_CNTL_3P3Z_CLA_DBUFF:n:+12, MR0
_ControlLaw_3P3Z_DBUFF_CLA_INIT_END:n:

.endm
```

B.4 CLA Code

```
.include "PeripheralAddress_ASM.h"
.cdecls   C,LIST,"LedDemo-CLAShared.h"
;// The following files have the assembly macros used in this file
.include "IIR2P2Z.asm"
.include "IIR3P3Z.asm"
;// As the macros are initialized by the C28x,
;// the net terminal being used by these macros are referenced here

.ref _CNTL_2P2Z_DBUFF1
.ref _CNTL_2P2Z_Coef1
.ref _CNTL_2P2Z_Fdbk1
.ref _CNTL_2P2Z_DBUFF2
.ref _CNTL_2P2Z_Coef2
.ref _CNTL_2P2Z_Fdbk2
.ref _CNTL_2P2Z_DBUFF4
.ref _CNTL_2P2Z_Coef4
.ref _CNTL_2P2Z_Fdbk4
.ref _CNTL_3P3Z_CLA_DBUFF5
.ref _CNTL_3P3Z_CLA_Coef5
.ref _CNTL_3P3Z_CLA_Fdbk5
CLA_DEBUG .set  0
.sect    "Cla1Prog"
.align  2

_Cla1Prog_Start:
;This task will Regulate Boost 1
_Cla1Task1:
    MNOP
    MNOP
    IIR2P2Z     1
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT1End:
;This task will Regulate Boost 2
_Cla1Task2:
    MNOP
    MNOP
    IIR2P2Z     2
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT2End:
;This task will Regulate Buck
_Cla1Task3:
    MNOP
    MNOP
    IIR2P2Z    4
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT3End:
;This task will Regulate Sepic
_Cla1Task4:
    MNOP
    MNOP
    IIR3P3Z   5       ; EPWM5A
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT4End:
;not used
_Cla1Task5:
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT5End:
;not used
_Cla1Task6: ;not used
    MSTOP
    MNOP
    MNOP
    MNOP
_ClaT6End:
;not used
_Cla1Task7:
    MDEBUGSTOP
    MSTOP
```

```
    MNOP
    MNOP
    MNOP
_ClaT7End:
;Used for initilization ,Only forced to run once from Main
_Cla1Task8:
    .if(CLA_DEBUG = 1)
        MDEBUGSTOP
    .endif
    IIR2P2Z_DBUFF_CLA_INIT 1
    IIR2P2Z_DBUFF_CLA_INIT 2
    IIR2P2Z_DBUFF_CLA_INIT 4
    IIR3P3Z_DBUFF_CLA_INIT 5
    MSTOP
_ClaT8End:

_Cla1Prog_End:
```

B.5 CLA ISR used for Dimming

```
#include "LedDemo-Settings.h"
#include "PeripheralHeaderIncludes.h"
#include "DSP2803x_EPwm.h"
Uint16 temp;
const int16 Kin= 65;
// Kin = Gfb * Gadc see Excel sheet
const int16 Kinb= 21;
// Kinb = (Gfb * Gadc)/10 see Excel sheet for Buck

interrupt void cla1_isr()
{
 EALLOW;
 PieCtrlRegs.PIEACK.all = PIEACK_GROUP11;  //Acknowledge the group containing CLA interrupts
//Check Boost1 OVP
    temp= ((long)AdcResult.ADCRESULT4 * (long)Kin)>>3;
    if( temp > Boost1_Threshold)
        {
        EPwm1Regs.TZFRC.bit.OST = 1;
        EPwm3Regs.ETSEL.bit.INTEN = 0;
        Boost1_OVF = 1;
        }
//Check Boost2 OVP
    temp= ((long)AdcResult.ADCRESULT5 * (long)Kin)>>3;
    if( temp > Boost2Temp)
       {
        EPwm2Regs.TZFRC.bit.OST = 1;
        Boost2_OVF = 1;
        EPwm6Regs.ETSEL.bit.INTEN = 0;
       }
//Check Buck OVP
    temp= ((long)AdcResult.ADCRESULT6 * (long)Kinb)>>2;
    if( temp > Buck_Threshold)
      {
       EPwm4Regs.TZFRC.bit.OST = 1;
         Buck_OVF = 1;
     EPwm3Regs.ETSEL.bit.INTEN = 0;
      }
// Check Sepic OVP
   temp= ((long)AdcResult.ADCRESULT7 * (long)Kin)>>3;

    if(temp > Sepic_Threshold)
      {
      EPwm5Regs.TZFRC.bit.OST = 1;
    Sepic_OVF = 1;
    EPwm6Regs.ETSEL.bit.INTEN = 0;
      }

EDIS;
AdcRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;      // Clear ADCINT1 flag reinitialize for next SOC
AdcRegs.ADCINTFLGCLR.bit.ADCINT2 = 1;     // Clear ADCINT2 flag reinitialize for next SOC
AdcRegs.ADCINTFLGCLR.bit.ADCINT3 = 1;     // Clear ADCINT2 flag reinitialize for next SOC
AdcRegs.ADCINTFLGCLR.bit.ADCINT4 = 1;     // Clear ADCINT2 flag reinitialize for next SOC
}
```

B.6 PWM Initialization

```
#include "PeripheralHeaderIncludes.h"
#include "DSP280x_EPWM_defines.h"
extern volatile struct EPWM_REGS *ePWM[];
void BuckSingle_CNF(int16 n, int16 period, int16 mode, int16 phase)
{
int16 temp;
(*ePWM[n]).TBCTL.bit.PRDLD = TB_IMMEDIATE;
(*ePWM[n]).TBPRD = period;
(*ePWM[n]).TBPHS.half.TBPHS = 0;
(*ePWM[n]).CMPA.half.CMPA = 0;
(*ePWM[n]).CMPB = 0;
(*ePWM[n]).TBCTR = 0;
(*ePWM[n]).TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
// set up-down mode for Symetric PWM
(*ePWM[n]).TBCTL.bit.HSPCLKDIV = TB_DIV1;
(*ePWM[n]).TBCTL.bit.CLKDIV = TB_DIV1;
temp = ( period * phase)/180;
// Converts Degrees to Clock Cycles
if(mode == 1) // config as a Master
{
(*ePWM[n]).TBCTL.bit.PHSEN = TB_DISABLE;
(*ePWM[n]).TBCTL.bit.SYNCOSEL = TB_CTR_ZERO;
// sync "down-stream"
}
if(mode == 0)
// config as a Slave (Note: Phase+2 value used to compensate for logic delay)
{
(*ePWM[n]).TBCTL.bit.PHSEN = TB_ENABLE;
(*ePWM[n]).TBCTL.bit.SYNCOSEL = TB_SYNC_IN;
if( 0 ≤ temp ≤ period)    // For  0 ≤ phase ≤ 180
{
(*ePWM[n]).TBPHS.half.TBPHS = temp+2;
(*ePWM[n]).TBCTL.bit.PHSDIR = TB_DOWN;
}
if(temp > period ) // For  180 ≤ phase ≤ 360
{
(*ePWM[n]).TBPHS.half.TBPHS = (temp+2-period);
(*ePWM[n]).TBCTL.bit.PHSDIR = TB_UP;
}
}

(*ePWM[n]).CMPCTL.bit.SHDWAMODE = CC_SHADOW;
(*ePWM[n]).CMPCTL.bit.LOADAMODE = CC_CTR_PRD;

(*ePWM[n]).AQCTLA.bit.ZRO = AQ_NO_ACTION;
(*ePWM[n]).AQCTLA.bit.CAU = AQ_CLEAR;
(*ePWM[n]).AQCTLA.bit.CAD = AQ_SET;
(*ePWM[n]).ETSEL.bit.SOCASEL = ET_CTRD_CMPB;
(*ePWM[n]).ETSEL.bit.SOCAEN = 1;
(*ePWM[n]).ETPS.bit.SOCAPRD = ET_1ST;
(*ePWM[n]).TZSEL.bit.OSHT1 = 1;
(*ePWM[n]).TZCTL.bit.TZA = 2;
//  Enable HiRes option
EALLOW;
(*ePWM[n]).HRCNFG.all = 0x0;
(*ePWM[n]).HRCNFG.bit.EDGMODE = HR_FEP;
(*ePWM[n]).HRCNFG.bit.CTLMODE = HR_CMP;
(*ePWM[n]).HRCNFG.bit.HRLOAD = HR_CTR_PRD;
EDIS;
// set CMPB to gererate Triger for ADC
(*ePWM[n]).CMPB = 5;
}
void Dimming_CNF(int16 n, Uint16 period)
{
int16 temp1;
(*ePWM[n]).TBCTL.bit.PRDLD = TB_IMMEDIATE;
(*ePWM[n]).TBPRD = period;
(*ePWM[n]).TBPHS.half.TBPHS = 0;
(*ePWM[n]).CMPA.half.CMPA = 0;
(*ePWM[n]).CMPB = 0;
(*ePWM[n]).TBCTR = 0;
(*ePWM[n]).TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
(*ePWM[n]).TBCTL.bit.HSPCLKDIV = TB_DIV1;
(*ePWM[n]).TBCTL.bit.CLKDIV = TB_DIV1;
(*ePWM[n]).TBCTL.bit.PHSEN = TB_DISABLE;
(*ePWM[n]).TBCTL.bit.SYNCOSEL = TB_SYNC_IN;
(*ePWM[n]).CMPCTL.bit.SHDWAMODE = CC_SHADOW;
```

```
(*ePWM[n]).CMPCTL.bit.SHDWBMODE = CC_SHADOW;
(*ePWM[n]).CMPCTL.bit.LOADAMODE = CC_CTR_PRD;
(*ePWM[n]).CMPCTL.bit.LOADBMODE = CC_CTR_PRD;
// NOTE-----------------------------------------------------
// due to sychronisation issue between the egdes of
// dimming pwm and triping of main PWM in ISR
// we will triger isr well before the edge
// Output A of this PWM module Control
(*ePWM[n]).AQCTLA.bit.ZRO = AQ_NO_ACTION;
(*ePWM[n]).AQCTLA.bit.CAU = AQ_CLEAR;
(*ePWM[n]).AQCTLA.bit.CAD = AQ_SET;
(*ePWM[n]).ETSEL.bit.INTSEL = ET_CTRU_CMPB;
//-----------------------------------------------------------
(*ePWM[n]).ETSEL.bit.INTEN = 0;
(*ePWM[n]).ETPS.bit.INTPRD = ET_1ST;
//Disable HiRes option
EALLOW;
(*ePWM[n]).HRCNFG.all = 0x0;
// set CMPA
temp1 = period >> 1;
(*ePWM[n]).CMPA.half.CMPA = temp1;   //    Set dimming PWM to 50% duty cycle
(*ePWM[n]).CMPB = temp1-5 ;
EDIS;
}
```

Appendix C

Source Code for Open Loop Measurement for Buck SMPS

C.1 CLA Code

Modified CLA code is listed below

```
;=====================================
IIR2P2Z_INIT    .macro n
;=====================================
; Variable Declarations
; The following are messages from the main CPU to the CLA  These are object pointer
_CNTL_2P2Z_Fdbk:n:  .usect "CpuToCla1MsgRAM",2  ; Feedback Pointer
_CNTL_2P2Z_Coef:n:  .usect "CpuToCla1MsgRAM",2  ; Pointer to filter Coefficients
;Publish Terminal Pointers for access from the C environment (optional)
.def    _CNTL_2P2Z_Fdbk:n:
.def    _CNTL_2P2Z_Coef:n:
; set terminal to point to ZeroNet
MOVL    XAR2, #ZeroNetCLA
MOVW    DP, #_CNTL_2P2Z_Coef:n:
MOVL    @_CNTL_2P2Z_Fdbk:n:, XAR2
MOVL    @_CNTL_2P2Z_Coef:n:, XAR2
.endm

;=====================================
IIR2P2Z     .macro n
;=====================================
_ControlLaw_2P2Z_Start:n:

MMOV16  MAR0, @_CNTL_2P2Z_Coef:n:   ;    MAR0  = points to Coeficient structure
MMOV16  MAR1, @_CNTL_2P2Z_Fdbk:n:   ;    MAR1  = points to feedback
MNOP
MNOP
MNOP
MMOV32      MR0, *MAR0[#2]++        ;    MR0   = scaling
MUI16TOF32 MR1, *MAR1               ;    MR1   = feedback  (ADC)
MMPYF32    MR1,MR1,MR0              ;    x     = MR1(feedback) - MR0(Qvolts)
MMOV32     MR0, *MAR0[#2]++         ;    MR0   = scaling
MADDF32    MR1,MR1,MR0
MMOV32     MR3, *MAR0[#2]++         ;    MR0   = Qvolts
MSUBF32    MR1,MR1,MR3
MUI16TOF32  MR3,  @_EPwm:n:Regs.TBPRD
MMINF32     MR1, #0.285             ;    Lower Limit
MMAXF32     MR1, #0.0               ;    Upper Limit
;////////////////////PWM Update Section  ////////////////////////////////////
MNOP
MNOP
MMPYF32     MR1, MR1, MR3
MMPYF32 MR1,MR1,#65536.0L    ; As CMPA is shifted by 16 bits
MF32TOI32   MR1, MR1
MMOV32  @_EPwm:n:Regs.CMPA.all, MR1
```

```
_ControlLaw_2P2Z_End:n:
.endm
```

C.2 Modifications in Main Code

Modification in main.c code is listed below

```
This File depitcs changes from orignal source code necssary to develop an understsand

struct inputDS2P2Z {
    float  scaling;          //  Scaling factor
    float  DC_bias;      //  DC operating point
    float  Qvolts;
    };

struct inputDS2P2Z Coef2P2Z_4;

Coef2P2Z_4.scaling=0.000032;     // Scaling Factor is found by imperical testing
Coef2P2Z_4.Qvolts=0.020928;    // DC point is chosen in continues conduction mode
Coef2P2Z_4.DC_bias=0.0;

CNTL_2P2Z_Fdbk4 = &AdcResult.ADCRESULT2;     // point to Buck output voltage value
CNTL_2P2Z_Coef4 = &Coef2P2Z_4;       // point to first coeff of Buck Loop
```