DIFFERENTIAL POWER PROCESSING FOR SERIES-STACKED PROCESSORS

BY

ANDREW R. STILLWELL

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Assistant Professor Robert Pilawa-Podgurski
ABSTRACT

The series-stacked architecture provides a method to increase power delivery efficiency to multiple processors. With a series-stack, differential power processing (DPP) is needed to ensure that processor voltages remain within design limits as the individual loads vary. This work demonstrates a switched-capacitor (SC) converter to balance a stack of four ARM® Cortex-A8 based embedded computers. A model of a series-stack with no DPP is first discussed for the case when loads can be controlled with no power electronics. We investigate hard-switched and resonant modes of operation in a ladder SC DPP converter, implemented with GaN transistors. Excellent 5 V regulation of each embedded computer is demonstrated in a 4-series-stack configuration, with realistic computational workloads. Moreover, we demonstrate hot-swapping of individual computers with maintained voltage regulation at all nodes. A peak stack power delivery of 99.8% is demonstrated, and DPP switching frequencies from 250 kHz to 2 MHz. Finally, the reliability of a series-stacked system is compared to an electrically parallel system.
To my wife, who provides the motivation, competition and inspiration.
ACKNOWLEDGMENTS

Special thanks to my parents, who got up way too early on Saturday mornings to drop me off at math competitions.

I’d like to thank Professor Sauer for help guiding me through my first semester at UIUC.

I’d like to thank the ECE department and the Grainger Foundation for funding my work.

Finally, I’d like to thank my advisor, Professor Pilawa-Podgurski, for taking me on so I didn’t have to go back to the “real world.”
TABLE OF CONTENTS

CHAPTER 1 BACKGROUND ............................................. 1
  1.1 Introduction .................................................. 1
  1.2 Series-Stack Model ........................................... 2
  1.3 Element-to-Element Optimal Order ......................... 5
  1.4 Voltage Regulation of a Series-Stack ..................... 11

CHAPTER 2 EXPERIMENTAL WORK ................................. 14
  2.1 Experimental Prototype ..................................... 14
  2.2 Experimental Results ....................................... 16
  2.3 Hot-Swapping ................................................ 24
  2.4 Series-Stacked Availability Modeling ..................... 26
  2.5 Conclusion .................................................. 39

APPENDIX A SUPPORT CODE ..................................... 41
  A.1 Series-Stacked Model ....................................... 41
  A.2 Model Test ................................................ 42
  A.3 PWM and Light Load Operation ........................... 46
  A.4 PWM Sweeping ............................................. 62

REFERENCES ....................................................... 70
CHAPTER 1
BACKGROUND

1.1 Introduction

Current trends in semiconductor scaling are driving down processor supply voltages to optimize performance in constrained power limits. In order to continue to improve system performance, suppliers have increasingly moved to parallelism. These two trends have led to increasing current levels, which are proving a challenge for typical 12 V to 1 V VRM applications that must process large currents. In systems today, all processors are connected electrically in parallel. Recently, it has been proposed to connect processor loads electrically in series so that the processors share the stack current [1], [2]. In order to regulate the voltages of the individual loads, differential power processing is applied to handle the mismatch currents. This series-stacked architecture allows the DPP circuitry to only process the mismatch current rather than the full current of the load. In the case of multiple processors with similar loads, this leads to low power processed and low power losses.

The series-stacked architecture has also been applied to multiple server applications [3-5] to reduce DC conversion power loss. The differential power processing circuitry for series-stacked systems could be implemented in many ways. Previous work has considered a “buck-boost” type converter similar to those used for battery charging [6], solar panel strings [7-9] and stacked digital circuits [10, 11]. Recent work has also considered the SC resonant ladder for solar PV strings [12]. Switched-capacitor converters offer high power density due to the higher energy density of capacitors compared to inductors. SC converters also have the potential to be implemented in CMOS [13], which would be beneficial in space constrained mobile applications.

This work describes a method to power a stack of four Texas Instruments BeagleBone Black ARM® based embedded computers. Each stacked com-
puter load consists of an AM335x 1GHz ARM Cortex-A8 processor with 512MB RAM and consumes up to 460 mA at 5 V. The ARM based computers were chosen because they feature a single core processor, extensive Linux support, and present a realistic load for such a system. When implementing a series-stack, start-up is a challenge. The stack should be balanced before any loads are connected or else a load may see an input voltage that is out of range. Also, an ideal stack would have provisions for hot-swapping out a load in the case of maintenance or outage. This work demonstrates one implementation that allows for a safe start-up and hot-swapping loads as needed. Finally, in cases where all loads are well matched, the power processed by the DPP converter is minimal and presents a light load condition. This thesis details a light load control scheme that improves converter efficiency at low power processed.

This work is the first to regulate the voltages on a series-stack of processors running application software. It also is the first to use a resonant ladder SC converter on a series-stack of active processor loads and actively hot-swap a processor. This work is organized as follows: Section 1.2 discusses a model of the behavior of the series-stack without DPP. When analyzing the series-stack with element-to-element DPP, it was found that the power processed by the stack is dependent on the order of the loads. Section 1.3 discusses the mathematical derivation for this behavior and formulates the optimal order into an equation. Section 1.4 discusses how differential power processing can be applied to a series-stack and Section 2.1 discusses a prototype (Figure 1.1) built based on this analysis. The experimental results taken from the prototype are discussed in Section 2.2. One of the most common concerns about series-stacked implementations is the reliability of the system since often one DPP failure will result in the failure of the entire stack of loads. Section 2.4 gives a brief analysis of a series-stacked converter in comparison with a parallel architecture. Finally, Section 2.5 presents the conclusion of this work.

1.2 Series-Stack Model

In an ideal case, a series-stack of loads would not need DPP. This section models and provides the behavior of a series-stack without DPP applied to
Figure 1.1: Block diagram of resonant ladder converter on a series-stack.

determine voltage swings given certain loads, and provides a foundation for future work. This model could also be used for simulation of a stack of controllable CPU loads. If the loads can be controlled by the processing tasks given to the individual processors in the stack, then the stack could be balanced over time even if it could not be balanced at a particular instant. Assuming that a processor or load can allow for some voltage swing in the supply voltage, the model shows how much the node voltages would swing in a particular imbalance. Figure 1.2 shows a simple stack of CPUs. Depending on the architecture and onboard power electronics, a processor can be modeled as a constant current, constant resistance or constant power load. In each case, the equations for solving for the node voltages are different. The following equations solve for the $k^{th}$ node voltage. Each relies on KCL and KVL of the stack.

Constant Current:

$$I_{CPU} = I$$  \hspace{1cm} (1.1)

$$I_{Stack} = I_{CPU} + I_C$$  \hspace{1cm} (1.2)

$$I_{Stack} = \sum_{i=1}^{n} \frac{I_{CPU_i}}{C_i}$$  \hspace{1cm} (1.3)

$$V_{ck} = \frac{I_{ck}}{C^k} t + V_{0k}$$  \hspace{1cm} (1.4)
Figure 1.2: A series-stack of CPUs with no DPP applied.

Constant Resistance:

$$I_{CPU} = \frac{V_k}{R_k}$$  \hspace{1cm} (1.5)

$$I_{Stack} = \frac{V_k}{R_k} + I_C$$ \hspace{1cm} (1.6)

$$I_{Stack} = \frac{V_{Stack} - \sum_{k=1}^{n} E_k V_{0k}}{\sum_{k=1}^{n} R_k (1 - E_k)}$$ \hspace{1cm} (1.7)

$$V_{ck} = V_{0k} E_k + (1 - E_k) I_{Stack} R_k$$ \hspace{1cm} (1.8)

$$E_k = e^{\frac{t}{R_k C_k}}$$ \hspace{1cm} (1.9)

Constant Power:

$$I_{CPU} = \frac{P_k}{V_k}$$ \hspace{1cm} (1.10)

$$I_{Stack} = \frac{P_k}{V_k} + I_C$$ \hspace{1cm} (1.11)
\[ I_{\text{Stack}} = \sum_{i=1}^{n} \frac{I_{\text{CPU}_i}}{C_i} \]  \hfill (1.12)

\[ V_{ck} = \text{ProductLog} \]  \hfill (1.13)

Equation 1.13 uses a second order Runge-Kutta approximation.

When applied to the same load mismatch, the behavior of each model becomes apparent. Figure 1.3 shows how each model calculates the node voltages when presented with the same loads, specifically \(I_1 = .9\) A, \(I_2 = .8\) A, \(I_3 = 1.2\) A and \(I_4 = 1.1\) A. The initial voltages were set at 1 V and the capacitors were set to 10 \(\mu\)F. In the case of constant current, the node voltage is linear. For constant resistance the node voltage is exponential, as expected of an RC circuit. The constant power model diverges as a logarithmic function. Figure 1.4 compares the node voltages directly. Note that all three models initially follow the same linear divergence from the balanced stack, and only diverge from each other over time. How quickly they diverge is a function of the load mismatch and capacitor values. Appendix Section A.1 contains the actual code.

1.3 Element-to-Element Optimal Order

Much as solar cells can be stacked in series to produce a large voltage for optimal power conversion, loads (specifically CPUs) can also be stacked to eliminate a conversion stage. The CPU series string faces a challenge similar to series solar cells in that the series current must be the same through all elements in the string, which is not the typical situation with CPUs processing varied loads. To compensate for this, differential power processors (DPP) can be applied to the series stacked CPUs. One topology of DPP is element-to-element as shown in Figure 1.5. These element-to-element converters transfer power/current between loads to allow individual CPUs to operate at their desired voltage and current while still being stacked in series with other loads.

While the element-to-element topology has the benefit of modularity and relatively low component stress, one downside is that the total power processed by the converters for N loads is dependent on the order/position of
the loads within the strings. This is evident in the derivations explained below. However, for stacked CPU loads a central controller can assign the load currents to desired locations within the stack to minimize the power processed and the power lost. This section seeks a closed form solution to
Figure 1.4: A comparison of the node voltages of the different models.

the order that minimizes the power processed, given a list of CPU currents to distribute.
Figure 1.5: CPU stack with element to element DPP.
1.3.1 The Problem

In this analysis, it is assumed that the CPU currents are known and that they can be assigned to any position within the stack. Given N CPU currents in a voltage balanced stack with ideal (100% efficient) DPP converters, the stack current, $I_{Stack}$, will just be the mean of the CPU currents. This can be derived from a power balance of the whole stack. From Figure 1.5 and KCL we can express the relationship of the DPP currents ($I_{di}, I_{Di}$) in terms of $I_{Stack}$ and CPU current ($I_{CPUi}$).

$$I_{di} - I_{Di-1} = I_{Stack} - I_{CPUi}$$  \hspace{1cm} (1.14)

Since the CPU voltages are matched throughout the string, and if it is assumed that power transfer between the two nodes is lossless, one can make the simplification that $I_{dn} = I_{Dn}$.

One can see that the mismatch from the stack current (the average of the CPU currents in the ideal case) determines the power processed so one can simplify the right half of the equation to just the mismatch:

$$I_{m,i} = I_{Stack} - I_{CPUi}$$  \hspace{1cm} (1.15)

The whole stack in terms of matrices takes the form:

$$
\begin{pmatrix}
1 & 0 & 0 & \cdots & 0 \\
-1 & 1 & 0 & \cdots & 0 \\
0 & -1 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & -1 & 1
\end{pmatrix}
\begin{pmatrix}
I_{d1} \\
I_{d2} \\
\vdots \\
I_{dn-2} \\
I_{dn-1}
\end{pmatrix}
= 
\begin{pmatrix}
\bar{I}_{m,1} \\
\bar{I}_{m,2} \\
\vdots \\
\bar{I}_{m,n-2} \\
\bar{I}_{m,n-1}
\end{pmatrix}
$$

Solving for the DPP currents:

$$
\begin{pmatrix}
I_{d1} \\
I_{d2} \\
\vdots \\
I_{dn-2} \\
I_{dn-1}
\end{pmatrix}
= 
\begin{pmatrix}
-1 & 0 & 0 & \cdots & 0 \\
-1 & -1 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & -1 & -1 & \cdots & 0 \\
-1 & -1 & -1 & \cdots & -1
\end{pmatrix}
\begin{pmatrix}
\bar{I}_{m,1} \\
\bar{I}_{m,2} \\
\vdots \\
\bar{I}_{m,n-2} \\
\bar{I}_{m,n-1}
\end{pmatrix}
$$

The power processed by DPP number $k$ is the CPU voltage, $V_k$, times the DPP current, $I_{dk}$. Since this is a voltage balanced stack every $V_k$ is the same:
\[ V = \frac{V_{\text{Stack}}}{N} \]. Note that the DPP current can be positive or negative but for this analysis we only care about the magnitude. From this, we can see that the power processed for each DPP is:

\[ P_k = V_k |I_{dk}| = V |\sum_{i=1}^{k} \bar{I}_{m,i}| \quad (1.16) \]

Total power is then:

\[ P_{\text{total}} = V \sum_{k=1}^{n-1} |\sum_{i=1}^{k} \bar{I}_{m,i}| \quad (1.17) \]

Finally, to bring this back in terms of given variables:

\[ P_{\text{total}} = V \sum_{k=1}^{n-1} |\sum_{i=1}^{k} \sum_{j=1}^{n} \frac{I_{CPU_j}}{n} - I_{CPU_i}| \quad (1.18) \]

Since the CPU currents are given, the problem thus boils down to finding the ordering of said currents to minimize the sum of the DPP currents, as given in Equation 1.18. While this can be done with a brute-force approach the question is whether an closed form solution can be obtained that is less computationally intensive. The brute force approach is an order N! problem to solve and very computationally intensive to solve for large stacks. No closed form solution was found in literature to solve for the optimal order, but two algorithms were investigated. The first was a “greedy” algorithm that started with a position in the stack, and found the optimal load for that position. It then moved on to each position until the stack was populated. The second was a less computationally complex “interleaved” algorithm that took advantage of the fact that if two loads of equal but opposite mismatches are placed side by side then they cancel out as far as the rest of the stack is concerned. It also took into account that the greatest mismatches should be placed in the middle of the stack so that they are balanced from above and below. The interleaved algorithm ordered the loads, and placed the extremes (largest and smallest loads) in the middle side by side. It then continued to work its way out on the stack, interleaving positive and negative mismatches. Figure 1.6 shows the results when simulating 100 random data sets of loads for a stack of 8 processor loads. As expected, the brute force approach to try every possible combination of loads results in the lowest average power processed, but the longest computation time. The greedy algorithm is slightly
better at finding the optimal order than the interleaved, but at two orders of magnitude more computation time. Finally, the unsorted result is by far the fastest, but at more than double the average power processed.

1.4 Voltage Regulation of a Series-Stack

1.4.1 Differential Power Processing

Traditional power architectures process all of the power delivered to a load. While many advancements have led to increased efficiency in the converters, the power losses still scale proportionally to the load. By electrically connecting loads in series, current through the loads is shared from load to load. However, if the current draws of all loads are not the same then the voltage at each node will drift. In order to regulate the voltage of each node, differential power processing can be applied to the stack. Since DPP only processes the mismatch in load current, the losses are now proportional to the current mismatches rather than the full system load. In the case of processors with similar loads, the current mismatches will be low, leading to greatly reduced power conversion loss and increased system efficiency. DPP architectures are typically classified as element-to-element (or load-to-load), bus-to-element (bus-to-load), or a hybrid of implementations of each [2]. For
this work the element-to-element architecture is applied in the form of a SC ladder converter.

1.4.2 Resonant Ladder SC Converter

The ladder converter is a popular SC topology because all capacitors and switches share the same low voltage rating, enabling high switching frequency with correspondingly small-sized passives, and high bandwidth control. When applied to a series-stack, the output capacitors at the load combine with the flying capacitors to form the SC network as shown in Figure 1.7. The capacitors and switches must be rated to the node voltage, in this case the input voltage range of the embedded computers (4.5 V-5.5 V). Each flying capacitor acts as the energy storage element for a differential power processor and the switches are shared to reduce component count. Resonant operation can be achieved with the addition of inductors in series with the flying capacitors. Resonant operation allows for zero current switching, which lowers switching losses and improves system efficiency. For a given flying capacitance, $C_f$, and a flying inductance, $L_f$, Equation 1.19 gives the resonant frequency. Note that this equation is derived assuming a constant voltage at each node. For a large current mismatch at a given node, this is not a valid assumption. However, with a sufficiently large output capacitance relative to the expected maximum current mismatch and the expected change during a switching period, the equation gives an adequate approximation for a switching period that will allow for zero current switching.

$$f_{sw} = \frac{1}{2\pi \sqrt{L_f C_f}} \quad (1.19)$$
Figure 1.7: Ladder converter schematic.
2.1 Experimental Prototype

Figure 2.1 shows an annotated photograph of the experimental prototype. The TI C2000 microcontroller generates the high resolution PWM waveforms for the two control signals and uses general purpose I/O signals to enable the soft-start/hot-swapping circuitry. Level shifters were used to power the gate

Figure 2.1: Experimental Prototype.
drivers for the GaN transistors of the converter and to transmit the isolated control signals. Level shifters were also used to drive the enable switches directly. Gate drivers were not used for the hot-swapping and enable switches since the enable switches are high-current, low-speed MOSFETs which do not require fast switching transitions. Table 2.1 provides a listing of the components used. For a stack of 4 embedded computers, the stack voltage was set at 20 V so that a balanced stack would supply 5 V to each load. Experimental measurements have been done in conventional and resonant modes of operation, with switching frequencies ranging from 250 kHz to 2 MHz.

The stack voltage and four node voltages were measured with the National Instruments PXIe-4300 data acquisition board. The stack current and four node currents were measured with 50 mΩ sense resistors, also with a NI PXIe-4300. All measurements were made with the 10 kHz input low-pass filter enabled and sampled at 20 kS/s. Current measurements were calibrated with an Agilent 34410A 6.5 digit DMM to ensure adequate accuracy in the high efficiency measurements. Figure 2.2 shows an annotated photograph of the test setup. For all test cases using the ARM® embedded computers, the HDMI ports, IO and USB ports were unused to ensure the processor
provided the main power draw.

Table 2.1: Component listing of the Resonant Ladder Converter

<table>
<thead>
<tr>
<th>Component</th>
<th>Part Number</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1 - S_8$</td>
<td>EPC 2014</td>
<td>40 V, 16 mΩ, 10 A</td>
</tr>
<tr>
<td>$C_{f1}, C_{f2}, C_{f3}$</td>
<td>CL10A106MQ8NNNC</td>
<td>6.3 V, 10 µF</td>
</tr>
<tr>
<td>$L_{f1}, L_{f2}, L_{f3}$</td>
<td>LQH31HN54NK03L</td>
<td>54 nH, 35 mΩ, 940 mA</td>
</tr>
<tr>
<td>$C_{out1} - C_{out4}$</td>
<td>CL31A476MPHNNNE</td>
<td>16 V, 47 µF</td>
</tr>
<tr>
<td>Gate drivers</td>
<td>LM5113</td>
<td>Half-bridge GaN driver</td>
</tr>
<tr>
<td>Gate resistors</td>
<td>ERJ-2GEJ1R1X</td>
<td>1.1 Ω</td>
</tr>
<tr>
<td>Gate driver Capacitors</td>
<td>GRM188R71A474KA61D</td>
<td>10 V, .47 µF,</td>
</tr>
<tr>
<td>Level-shifting</td>
<td>ADUM5210</td>
<td></td>
</tr>
<tr>
<td>Micro-controller</td>
<td>TMS320F28035</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Experimental Results

2.2.1 Resonant Operation

The component values for flying capacitance and inductance were used with Equation 1.19 to find an initial estimate for resonant frequency operation on the prototype. It is well known that ceramic capacitors operating under a DC bias will suffer from a derating in capacitance, which leads to an increase in resonant frequency. In addition, parasitic inductance from PCB traces leads to a decrease in resonant frequency. Due to these component characteristics and parasitics, the resonant frequency was found experimentally to be 250 kHz. Figure 2.3 shows the inductor current in the case of CPU$_1$ and CPU$_2$ at 100% CPU utilization using the Linux “stress” utility [14] and CPU$_3$ and CPU$_4$ idle.

For this system, there are two efficiency metrics that characterize the converter performance. The system or “stack” efficiency is the ratio of the power delivered to the loads ($P_{out}$, which represents the sum of all computer input power) to the power supplied to the system ($P_{in}$, drawn from the supply), given in Equation 2.1:

$$\eta_{stack} = \frac{P_{out}}{P_{in}}$$ (2.1)
As the DPP converter only processes the mismatch in load currents, it is not directly proportional to the power supplied to the loads. Therefore the converter efficiency is a measure of how efficiently the converter processes the mismatch power, given in Equation 2.2:

$$\eta_{\text{converter}} = \frac{P_{\text{processed}} - P_{\text{loss}}}{P_{\text{processed}}}$$ (2.2)

Four embedded computers were stacked in series and balanced by the converter. With the network port connected but the processor idle on all boards, all node voltages were regulated to within 44 mV of 5 V. Table 2.2 shows the performance metrics for this case as Case 1. Note that the converter efficiency is relatively low at 89.7% for this operating mode, which can be considered an extreme light load case. However, in this case the loads are well balanced, and the converter does not process much power. This results in a total stack efficiency of 99.8%. Next, CPU1 and CPU2 were set to 100% CPU utilization with the Linux ‘stress’ utility while CPU3 and CPU4 were at idle (Case 2). In this case, the converter efficiency is higher because it is
processing more power, but the stack efficiency is decreased to 98.7% because more total power is lost in processing. With a greater load mismatch, the node voltages were still held within 90 mV in this scenario. The converter was designed to handle the full mismatch currents of a stack of embedded computers so the final test case (Case 3) was to power a single processor. The node voltage at the powered load ($CPU_1$) was held within 80 mV of 5 V. It should be noted that this scenario represents a poor use case for the series-stack architecture, and for which the converter was not optimized. Even so, the converter still runs at a 94.1% efficiency and the stack efficiency is 92.9%. The efficiency of the stack is lower than the converter efficiency because in this case the converter processes more power than the stack delivers to the loads, as seen in Case 3 of Table 2.2. This is not unexpected in the element-to-element architecture when the load mismatches are at extremes [2].

Table 2.2: Converter Performance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{out}$ (W)</td>
<td>5.02</td>
<td>5.35</td>
<td>1.28</td>
</tr>
<tr>
<td>$P_{loss}$ (W)</td>
<td>8.8m</td>
<td>89.7m</td>
<td>102m</td>
</tr>
<tr>
<td>$P_{processed}$ (W)</td>
<td>85.8m</td>
<td>1.99</td>
<td>1.71</td>
</tr>
<tr>
<td>Converter $\eta$</td>
<td>89.7%</td>
<td>95.5%</td>
<td>94.1%</td>
</tr>
<tr>
<td>Stack $\eta$</td>
<td>99.8%</td>
<td>98.7%</td>
<td>92.9%</td>
</tr>
</tbody>
</table>

To generate a plot of the converter efficiency, three loads were held constant at the same load while the fourth was swept from minimal mismatch to a large mismatch. The power processed varied with the mismatch generated by the fourth load. Results are shown in Figure 2.4 with a peak efficiency of 98.7%. Note that this is the efficiency of the DPP power conversion (power loss vs power processed) rather than the efficiency of the stack (power loss vs power delivered to the loads).

Electronic loads were used to induce current steps to simulate CPU load transients. Figure 2.5 shows the recorded node voltages in response to the load currents (Figure 2.6). Initially, all loads are relatively well matched and thus the node voltages are well balanced. Just before the 10 second mark, the current in load 2 ($I_2$) drops from above 300 mA to just above 200 mA. This creates a mismatch, which the converter handles, and keeps the stack node voltages balanced to within 70 mV. Note that the node voltage, $V_2$, slightly increased with the drop in current. This is because the drop in current is
Figure 2.4: Converter efficiency vs power processed.

equivalent to a rise in load input impedance. This impedance is in series with the other loads and the stack behaves like a voltage divider. As the impedance at the lightly loaded CPU is higher than the other loads, the node voltage is also slightly higher. The DPP converter works to reduce this mismatch but is not completely successful because of switching losses. At the 15 second mark, $I_2$ returns to the previous load and the stack rebalances. At the 20 second mark, $I_3$ steps up to above 400 mA and the node voltages again change. In this case, the impedance of load 3 is the least, so it experiences the lowest node voltage. In both cases, the converter handles the current steps without oscillation on the outputs within the measured frequency bandwidth (10 kHz).

2.2.2 Light Load Operation

While the DPP converter must be able to handle a suitable mismatch load, in cases where stacked processors are all processing similar loads mismatch currents are low and DPP is not needed. Matched loads introduce the need
for light load operation of the DPP to further improve the efficiency of the system. To detect a light load condition, the controller must determine whether the loads are well balanced. One way to do this would be a current sense measurement of each load current. However, most practical current sense measurements for currents at this level are lossy and would lead to losses proportional to the power process. Another option is to measure just the mismatch currents, again introducing losses but only proportional to the mismatch currents. Both methods would require amplified current sense measurements level-shifted to the node stack voltage.

Rather than add this additional complexity, the scaled node voltages were measured with the C2000 onboard ADC. As noted previously, under heavy load the converter is not able to perfectly regulate the node voltages but under light load the node voltages are well regulated. By sensing the node voltages the controller is thus able to determine whether to enter light load operation. In light load operation, the switching frequency was reduced by 10x of the resonant frequency to 25 kHz. This reduction in switching frequency reduces switching losses and further improves efficiency at light loads.
Hysteresis control was applied as follows: if all node voltages were within the hysteresis limit of the average of the node voltages, then the controller went into light load operation. If any of the node voltages were outside of the hysteresis limit then the controller went back into resonant operation. Appendix Section A.3 contains the code on the C2000 that implements the light load control.

Figure 2.7 shows the PWM control signals and the resulting inductor currents of $I_{L3}$ with 4 embedded PCs operating at idle. In light load operation the switching frequency is lower, and the inductor current is not resonant but does show the expected damped LC ringing. When the converter engages resonant mode the resonant current increases in magnitude as the node voltages are rebalanced before settling back into resonant steady state operation. Figure 2.8 shows the results with the same loads over a longer period of time. In this case, the controller leaves light load for only a short period of time before the nodes are rebalanced and the controller re-enters light load operation.

As mentioned, light load operation should improve efficiency in cases when loads are well matched. Figure 2.9 shows the converter efficiency vs. power...
Light Load Operation Resonant Operation

Figure 2.7: In light load operation, switching no longer occurs at the zero crossing.

processed with light load operation applied for low power processed. For power processed less than 0.5 W, light load operation is shown to increase efficiency over resonant operation.

Thus far efficiency plots have not taken into account the power required to drive the LM5113 gate drivers. As this power is a function of switching frequency and light load operation reduces the average switching frequency, the gate driver power was recorded for light load and resonant operation modes. Figure 2.10 shows the measured power to the LM5113 gate drivers for various power loads with resonant operation and with light load control enabled. Since the switching frequency was constant in resonant operation, the power to the gate drivers was constant relative to the power processed. With light load control enabled, at lower power processed the controller engaged light load operation heavily, which led to lower average switching frequency and lower gate driver power. As the power processed increased, the controller spent less time in light load operation and the gating power increased until it converged with the resonant operation. This convergence was expected since
Figure 2.8: With balanced loads the converter is only briefly in resonant operation.

the node mismatch was large enough that the controller would never engage light load mode.

Figure 2.11 shows the efficiency of the converter with the gating losses included. Light load operation still improved the converter efficiency at low power processed. Even with gating losses included, the converter reached a peak efficiency of 97% and a peak stack efficiency of 99.2%.

2.2.3 Hard-Switched Operation

The flying inductors were removed to evaluate the converter in non-resonant operation. The flying capacitors were kept at the same values for this comparison. The switching frequency was swept and the power losses were measured to determine an optimal switching frequency. As shown in Figure 2.12, the ladder converter operated in the slow switching limit [15] up to 100 kHz after which it entered the fast switching limit. The converter was successfully operated up to 2 MHz but the efficiency suffered and the switching losses in-
Figure 2.9: Light load operation improves the converter efficiency when loads are well balanced.

creased. Appendix Section A.4 contains the code that sweeps the switching frequency.

The converter was set to a fixed switching frequency of 200 kHz and the procedure used to characterize the resonant operation was applied to the hard switched converter. Figure 2.13 compares the resonant operation efficiency to the non-resonant efficiency across power levels.

2.3 Hot-Swapping

Figure 2.14 shows the hot-swap circuitry implemented for this converter. The circuit implemented a “high impedance” path that is enabled first. The high impedance path provides a “soft-start” for the load and limits any input current pulses resulting from load capacitance. This protects the load and prevents the converter node voltages from becoming out of balance. The impedance is provided in the form of a series resistor, $R_{\text{limit}}$. After the load capacitances have charged up, the high impedance path is no longer desired,
Figure 2.10: The gate driver power is reduced in light load operation.

so a low impedance path is enabled. The delay to the low impedance path is provided by a simple RC filter formed by $R_D$ and $C_D$ shown in Figure 2.14. The diode is provided to ensure that turn-off is not similarly delayed. For these tests, $R_D$ and $C_D$ were set to 30 kΩ and 10 $\mu$F respectively to form an RC time constant of 300 mS. The large delay was introduced not only to charge up the input capacitors but to account for the power-up delay incorporated in the PMIC on the embedded computer.

The hot-swap circuitry was tested by removing a single embedded computer from a balanced stack. Figure 2.15 shows the node voltages and Figure 2.16 shows the node currents during a hot-swap event. At the beginning of the measurement, all embedded computers are connected to the network but idle, resulting in a well balanced stack. At the 3 second mark, $CPU_2$ is disconnected from the stack via the hot-swapping circuitry. Note that in this case, the node voltages were measured at the converter side so the node voltages show $V_2$ still within limits, but the voltage to $CPU_2$ is 0 V. The series-stack sees a high impedance at Node 2 so it has the highest voltage, but still within the safe operating range of the embedded computer. At
the 5 second mark $CPU_2$ is reconnected with a soft start and goes through the boot process. The voltage mismatch is reduced and the node voltages further rebalance.

2.4 Series-Stacked Availability Modeling

In an effort to increase the power efficiency of servers, some manufacturers are moving to multi-processor solutions (such as the HP Moonshot servers [16]). For many, small, highly parallel operations, these servers can provide better performance with high energy efficiency when compared to traditional servers with a single large CPU. Currently these multiple processors are powered with traditional power electronics, typically a switching step-down converter in which all of the power to the CPU must flow through the converter. In this case, the power loss of the system is proportional to the CPU power. As the CPU power increases, so too does the power loss. By electrically connecting multiple processors in series, one can power multiple processors
while providing a voltage step-down to meet the input voltage specifications of the processors [1]. In the case of mismatched loads, differential power processing (DPP) can be applied to route the mismatch current and regulate the node voltages. In the case of DPP, the power loss is proportional to the mismatch (from other CPUs in the stack) in CPU power rather than the CPU power. This means that if the CPUs are all well matched in power load, very little power is processed and lost. This mechanism allows for very high conversion efficiency even when supplying high power to multiple processors.

While there have been many applications of DPP on a series stack [1,2,7,12,13] none have performed a detailed analysis on the reliability of the architecture compared to a traditional parallel architecture. Figure 2.17 shows the typical parallel case for four CPUs. Each CPU is powered from a DC-DC converter in parallel. For reliability analysis, each DC-DC converter will be considered a component. A system failure will occur when all four DC-DC converters are down at the same time. Each can be repaired independently.

Figure 2.18 shows a series-stack of four CPUs with a load-to-load [2] DPP
Figure 2.13: Efficiency of light load and resonant operation with gate drive losses included.

Figure 2.14: Hotswap circuitry for $CPU_N$. 
structure. Only three DPP converters (specifically, $DPP_1$, $DPP_2$, $DPP_3$) are needed to balance a stack of four CPUs. Each can be considered a component with a failure rate. In the case of three DPPs, a single component failure will result in unsuitable voltage regulation to the CPUs, so a single failure will result in a system failure. However, the addition of a single DPP, $DPP_4$ in Figure 2.18, adds a level of redundancy such that two DPPs must fail before the system fails. This is considered a hybrid DPP. It should be noted that in this system, it is not feasible to repair a DPP while the system is still in operation so no repairs will occur until two DPPs have failed.

This section uses Markov models to compare the availability of the parallel, load-to-load, and hybrid systems. Since reliability data are not readily available for the different converters, relative values are used with sensitivity analysis added to show the impact of these values. In addition to considering the mean time to failure (MTTF) for the system, the analysis considers the average downtime for the four processors in a ten year period. This metric will give a more relevant comparison of the system architectures and allow for an analysis of the failure rates that lead to equal downtime for the different
systems.

2.4.1 Markov Availability Models

The parallel, load-to-load and hybrid architectures result in significantly different Markov availability models. Table 2.3 summarizes the failure and repair rates for each system.
2.4.2 Parallel Power Supplies

The traditional parallel architecture (Figure 2.17) can be modeled as four parallel components since computation still takes place unless all four converters are offline at the same time. For the purpose of this analysis, a best in class 48 V SynQor PQ60120QEx25 48 V to 12 V DC-DC converter will be used as the comparison. The SynQor converter specifies [17] a mean time before failure (MTBF) of $10^6$ h leading to a failure rate ($\lambda$) of $10^{-6}$ failures/hour. This failure rate is extremely low and, when the original calculations were made, led to extremely high reliability. For the purposes of this analysis, the failure rate was adjusted to $10 \times 10^{-6}$ h. While a repair rate is not specified, in a practical case the failing server blade would just be removed from the rack and replaced with a new functioning unit while
the failed unit would be serviced off line. For the purposes of this analysis, the repair rate ($\mu$) is fixed at 1 repair/h. It is assumed that if multiple components have failed, they are all repaired. The incremental time to repair multiple servers is considered negligible because whether repairing one server or four, most of the repair time is consumed by dispatching a technician to perform the repair. Figure 2.19 shows the reduced Markov availability model of the parallel architecture.

2.4.3 Load-to-Load Differential Power Processing

In the load-to-load structure (Figure 2.18), if a single DPP converter fails, the node voltage is not guaranteed to operate within the safe operating range of the processor. In order to regulate the voltage at a node, a DPP must be present to conduct mismatch current around a particular node. For example, if $DPP_1$ fails, the only path for the stack current is through $CPU_1$, so the voltage at that node cannot be regulated. Therefore, the converters can be modeled as series connected components for the sake of reliability analysis. It should be noted that for a stack of $N$ nodes, only $N - 1$ DPP converters are needed to regulate the stack. Figure 2.20 shows the Markov availability model state-transition diagram. There are no commercial power supplies specifically designed for DPP operations, but the complexity of a typical DPP

<table>
<thead>
<tr>
<th>System</th>
<th>$\lambda$ ($h^{-1}$)</th>
<th>$\mu$ ($h^{-1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>$10 \times 10^{-6}$</td>
<td>1</td>
</tr>
<tr>
<td>Load-to-load</td>
<td>$10 \times 10^{-6}$</td>
<td>1</td>
</tr>
<tr>
<td>Hybrid</td>
<td>$10 \times 10^{-6}$</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.19: Markov availability model state-transition diagram for parallel power supplies.
converter is similar to a traditional switching power converter. In addition, for the purposes of start-up and extreme mismatches, DPP converters should be rated to handle full (or potentially double) current of a single processor but for the majority of their operation will be processing a much smaller percentage of the rated current. This means that a DPP converter could not only share failure rates with a similar parallel converter, but in typical conditions surpass them. For the case of this analysis, the DPP failure rate is considered equal to the comparison converter, with sensitivity analysis included later.

![Markov availability model state-transition diagram for the load-to-load DPP structure.](image_url)

**Figure 2.20:** Markov availability model state-transition diagram for the load-to-load DPP structure.

### 2.4.4 Hybrid Differential Power Processing

The hybrid architecture adds one level of redundancy at the cost of an additional DPP converter. As shown in Figure 2.18, this redundancy is achieved by adding $DPP_4$ that regulates the voltage between top and bottom halves of the stack. In the case mentioned previously, if $DPP_1$ were to fail, the mismatch current from $CPU_1$ can be processed by $DPP_4$. At first glance, it appears that $DPP_1$ and $DPP_3$ can fail with $DPP_4$ handling the mismatch current of each, but in addition to handling the mismatch current the DPP basically matches the voltages between the two nodes it services. Without $DPP_1$ or $DPP_3$, there is no mechanism to ensure that the voltages at nodes 1 and 4 match the voltages at nodes 2 and 3. It should be noted that $DPP_4$ in Figure 2.18 has to handle double the voltage of the other DPP converters. However, for this thesis it is assumed that this converter has the same failure rate as the other converters. This assumption is made because for practical
purposes, the components that make up DPP1, DPP2, and DPP3 would be rated at higher than the node voltage for the case of start-up or shorts. For the failure case, it would not be practical to repair a DPP while the stack is still operating, nor is this study considering detection of a single DPP failure. Therefore, no repairs will occur until two DPP converters have failed, causing the system to fail. Figure 2.21 shows the state-transition diagram for the hybrid DPP architecture.

Figure 2.21: Markov availability model state-transition diagram for the hybrid DPP structure.

Equation 2.3 gives the Chapman-Kolmogorov equations for the parallel architecture, where \( \pi_i(t) \) gives the probability of the system being in state \( i \) at time \( t \).

\[
\begin{bmatrix}
    \dot{\pi}_1(t) \\
    \dot{\pi}_2(t) \\
    \dot{\pi}_3(t) \\
    \dot{\pi}_4(t) \\
    \dot{\pi}_5(t)
\end{bmatrix}^T =
\begin{bmatrix}
    \pi_0(t) \\
    \pi_1(t) \\
    \pi_2(t) \\
    \pi_3(t) \\
    \pi_4(t)
\end{bmatrix}^T \times
\begin{bmatrix}
    -4\lambda & 4\lambda & 0 & 0 & 0 \\
    \mu & -(3\lambda + \mu) & 3\lambda & 0 & 0 \\
    \mu & 0 & -(2\lambda + \mu) & 2\lambda & 0 \\
    \mu & 0 & 0 & -(\lambda + \mu) & \lambda \\
    \mu & 0 & 0 & 0 & -\mu
\end{bmatrix}
\]

(2.3)

Equation 2.4 gives the Chapman-Kolmogorov equations for the load-to-load architecture:

\[
\begin{bmatrix}
    \dot{\pi}_1(t) \\
    \dot{\pi}_2(t)
\end{bmatrix}^T =
\begin{bmatrix}
    \pi_1(t) \\
    \pi_2(t)
\end{bmatrix}^T \times
\begin{bmatrix}
    -3\lambda & 3\lambda \\
    \mu & -\mu
\end{bmatrix}
\]

(2.4)

Equation 2.4 gives the Chapman-Kolmogorov equations for the hybrid architecture:
\[
\begin{bmatrix}
\dot{\pi}_1(t) \\
\dot{\pi}_2(t) \\
\dot{\pi}_3(t)
\end{bmatrix}^T = 
\begin{bmatrix}
\pi_1(t) \\
\pi_2(t) \\
\pi_3(t)
\end{bmatrix}^T \times 
\begin{bmatrix}
-4\lambda & 4\lambda & 0 \\
0 & -3\lambda & 3\lambda \\
\mu & 0 & -\mu
\end{bmatrix}
\]  
(2.5)

2.4.5 Availability Results

Matlab was used to solve the Chapman-Kolmogorov equations given above for the three different architectures. The long-term probabilities for each state in the parallel case are given in Table 2.4. As States 1-4 represent operational states, the long term reliability of this system is virtually 1 (1 − 2.4 × 10\(^{-19}\) to be exact).

Table 2.4: Long-Term Probabilities of the Parallel Structure

<table>
<thead>
<tr>
<th>State</th>
<th>Long-term probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\pi_1 = 0.999960001599936)</td>
</tr>
<tr>
<td>2</td>
<td>(\pi_2 = 0.000039997200148)</td>
</tr>
<tr>
<td>3</td>
<td>(\pi_3 = 0.000000001199892)</td>
</tr>
<tr>
<td>4</td>
<td>(\pi_4 = 0.000000000000024)</td>
</tr>
<tr>
<td>5</td>
<td>(\pi_5 = 2.4 \times 10^{-19})</td>
</tr>
</tbody>
</table>

Table 2.5 shows the long-term probabilities for the states in the load-to-load architecture. As there are only three converters to fail rather than four in the cases of the parallel and hybrid structures, the probability of no failures (\(\pi_1\)) is the highest for this case, but as one component failure results in a system failure the long term availability is the lowest at .99997.

Table 2.5: Long-Term Probabilities of the Load-to-Load DPP Structure

<table>
<thead>
<tr>
<th>State</th>
<th>Long-term probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\pi_1 = 0.999970000899973)</td>
</tr>
<tr>
<td>2</td>
<td>(\pi_2 = 0.000029999100027)</td>
</tr>
</tbody>
</table>

The addition of one level of redundancy but no repairs while operating has an interesting effect on the long-term probabilities, as shown in Table 2.6. Since repairs do not occur when just a single component has failed, the long term probability that the system is in a state with a single component failure (\(\pi_2\)) is actually higher than that with no component failures. Also, the one
level of redundancy leads to a slightly higher availability (0.99998) than the load-to-load case.

Table 2.6: Long-Term Probabilities of the Hybrid DPP Structure

<table>
<thead>
<tr>
<th>State</th>
<th>Long-term probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\pi_1 = 0.428564081758598$</td>
</tr>
<tr>
<td>2</td>
<td>$\pi_2 = 0.571418775678131$</td>
</tr>
<tr>
<td>3</td>
<td>$\pi_3 = 0.000017142563270$</td>
</tr>
</tbody>
</table>

The mean time to failure (MTTF) of each architecture is given in Table 2.7. Note that the parallel architecture MTTF is high enough that in reality the failure rate could no longer be considered constant and would be better modeled as a function decreasing with time. This analysis is not included in this thesis.

Table 2.7: MTTF for Each Configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>MTTF (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>$4.24 \times 10^{18}$</td>
</tr>
<tr>
<td>Load-to-load</td>
<td>$3.33 \times 10^{4}$</td>
</tr>
<tr>
<td>Hybrid</td>
<td>$5.83 \times 10^{4}$</td>
</tr>
</tbody>
</table>

A more interesting metric is to look at the total number of down hours over a ten year period. In the parallel case, recall that the system is considered operational if a single processor is operational, so some time may be spent with one or more processors nonoperational. For example, in State 2 only three processors are operational and in State 3 only two are operational but both states are considered operating states in this analysis. For both the load-to-load and hybrid cases, all processors are operational or none are. Equations 2.6, 2.7, and 2.8 give the number of down hours for the parallel, load-to-load, and hybrid cases respectively:

\[
down_{\text{par}} = 87660(4 - (4\pi_1 + 3\pi_2 + 2\pi_3 + \pi_4)) \quad (2.6)
\]

\[
down_{\text{load}} = 87660(4 - 4\pi_1) \quad (2.7)
\]

\[
down_{\text{hybrid}} = 87660(4 - (4\pi_1 + 4\pi_2)) \quad (2.8)
\]
Each equation is derived by considering the number of processors operational in each state and the probability of the system being in that state.

Table 2.8 shows the results for downtime over a ten year period. This comparison is much more favorable to the DPP architectures. While the parallel architecture virtually never has all four processors offline at the same time, it does spend time with one or more processors down.

Table 2.8: Downtime for Four Processors Operating over 10 Years

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Downtime (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>3.5064</td>
</tr>
<tr>
<td>Load-to-load</td>
<td>10.5189</td>
</tr>
<tr>
<td>Hybrid</td>
<td>6.0109</td>
</tr>
</tbody>
</table>

As stated earlier, as the DPP converters are likely to be overrated for the typical current that they will be processing, it is reasonable to suggest that the DPP converters’ failure rate can be lower than that of a traditional power supply. To determine to what extent the failure rate needs to improve, the failure rate was swept from $10 \times 10^{-6}$ to $10 \times 10^{-7}$. Figure 2.22 shows the results plotted with the constant downtime of the parallel configuration.

As can be seen, the relationship between the DPP failure rate and downtime is linear. Using this relationship, the failure rates at which the downtimes for all systems are the same can be found. These results are shown in Table 2.9. Basically the failure rate of the DPP would need to be one third of the failure rate of the parallel converter in the load-to-load case and less than half for that of the hybrid case.

Table 2.9: Failure Rates that Lead to Equal Downtime.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Failure Rate ($h^{-1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>$10 \times 10^{-6}$</td>
</tr>
<tr>
<td>Load-to-load</td>
<td>$3.33 \times 10^{-6}$</td>
</tr>
<tr>
<td>Hybrid</td>
<td>$5.83 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

Generally speaking, this requirement is within reason as thermal breakdown is a known detriment to long-term reliability. Since the DPP converters are processing less power, they would have less power loss and less thermal dissipation than a parallel converter of the same efficiency. However, at the already low failure rates of the reference supply it becomes harder to make
that argument. It should be noted that this ratio of the failure rates between the DPP converters and parallel converter that leads to the same downtime is constant over failure rates. In other words, if you compare the DPPs to less reliable parallel converters, the DPP converters still need to be one third and less than one half for the load-to-load and hybrid respectively.

2.4.6 Availability Future Work

This analysis has focused on the availability of the systems, but future work should incorporate reliability, which was excluded due to space constraints. For a stack of four processors, only a single DPP is needed for one additional level of redundancy, but for larger stacks this would not be the most likely implementation. In order to keep the voltage ratings down the most likely scenario would be to add an additional hybrid DPP for each additional pair of processors added. In other words, in addition to the \( N - 1 \) load-to-load DPP converters, one would want \( N/2 - 1 \) hybrid DPP converters. The interesting
part of this is that depending on which DPPs failed, it could take more than two component failures to result in the system failure. The complexity of the Markov state transition diagram would increase proportionally and could yield some interesting results.

2.4.7 Availability Conclusion

This section has provided a Markov availability analysis of three methods to power four processors. The traditional parallel approach processes all of the power delivered to the loads but exhibits the highest MTTF and availability. The load-to-load DPP architecture only processes the load mismatch and thus may have a lower failure rate but, at the same failure rate as the parallel case, exhibits the lowest MTTF and availability. Adding one DPP results in a hybrid DPP structure with one level of redundancy. This hybrid DPP approach retains the load-to-load benefits of power processing and increases the MTTF and availability, though not up to the levels of the parallel structure. In both DPP cases, if the failure rates of the DPP converters can be shown to be decreased by one third (load-to-load) or less than half (hybrid) of the parallel converters failure rate, then the availability of the four processors is equalized.

2.5 Conclusion

This work presented a resonant ladder SC converter that performed differential power processing on a series-stack of four embedded computers. A model of a series-stack with no DPP was developed and discussed with respect to constant current, resistance and power modes. The benefits of DPP, and specifically of a resonant ladder converter, were analyzed. A prototype was built using GaN transistors controlled by a TI C2000 microcontroller. Four embedded computers were powered in a stack and regulated by the resonant ladder converter with a peak conversion efficiency of 98.1% and a peak stack efficiency of 99.8%. Hard switching operation was evaluated compared to resonant operation and found to be less efficient in this converter. Current steps were induced in loads and the resulting node voltage responses were discussed. Light load operation was included through hysteresis control and
successfully demonstrated efficiency improvements at lower power mismatch. In addition, an embedded computer was actively hot-swapped from a stack and successfully reinserted with a soft-start. Finally, the availability of a series-stacked system was compared to that of a parallel system and it was found that if the failure rate of the DPP used in a series-stack is half that of the parallel system, the total downtime in a 10 year period is the same.
A.1 Series-Stacked Model

This function calculates an array of stack voltages \( V \), given a number of loads \( (N) \), stack voltage \((V_{\text{stack}})\), array of loads \((\text{Load})\), array of CPU capacitors \((C_{\text{CPU}})\), array of initial voltages \((V0)\), and the load model \((CPU\text{Model})\). The CPU model (‘current’, ‘resistance’, or ‘dcdc’) determines the units of the loads.

```matlab
function V = Series_Model(N, Vstack, Load, Ccpu,t, V0, CPU_Model)
%Initial Calculations

if (~exist('V0'))
    V0 = ones(1,N)*Vstack/N; %Initial voltages if none given, assumes evenly distributed load
end

if (~exist('CPU_Model'))
    CPU_Model = 'current'; %use current model if none given
end

switch CPU_Model
    case 'current' %constant current model for a CPU
        %String current - weighted by the caps, see derivation on page 2 of %notebook
        IStack = sum(Load./Ccpu)/sum(ones(1,N)./Ccpu);
        Ic = IStack - Load; %capacitor current
```
V = Ic./Ccpu*t + V0; %assuming Ic is constant, Vc changes linearly over time

case 'resistance' %constant resistance model for a CPU
%see derivation in notebook page 16-20
E = exp((-t*ones(1,N))./Ccpu./Load);
IStack = (Vstack - sum(E.*V0))/sum(Load.*(1-E));
V = (V0 - IStack*Load).*E +IStack*Load;

case 'dcdc' %DC/DC load - basically constant power
%since dv/dt = -P/(CV) + Istack/C, this is a non-linear ODE
%the solution is a Wronskian/Product Log... no fun to work with
%therefore using a 2nd order Runge-Kutta approximation with a time step of tStep - this may need to be raised/lowered based on
%accuracy/computation needs. Could also consider Adams-Bashforth
    tStep = 1e-9; %time step determines accuracy, might need to lower this
    V = V0;
    for i=1:floor(t/tStep)
        IStack = sum(Load./V./Ccpu)/sum(ones(1,N)./Ccpu);
        f = -Load./Ccpu./V + IStack./Ccpu;
        k1 = tStep*f;
        k2 = tStep*(-Load./Ccpu./(V+k1) + IStack./Ccpu);
        V = V + (k1+k2)/2;
    end

otherwise
    warning('Invalid CPU model declared in Series_Model');
end
end

A.2 Model Test

This code generates plots of the three models with a given mismatch. Uses the Series-stacked model function above.
%model of a bunch of CPUs (Current sources in parallel with caps) 
in series

clear; clc

%Inputs
Vstack = 4; %stack voltage in volts
N = 4; %number of processors
t = .5e-6; %timestep
Icpu1 = [.5 .6 .4 .5]; %N CPU load in amps - should this be in PU?
Icpu2 = [1 .6 .4 .5];
Icpu3 = [.5 .6 .4 .5];
C = 1e-6*[10,11,12,13];

V0 = [1,1,1,1];

Vplot1 = zeros(4,N);
Vplot1(1,:) = V0;
V0 = Series_Model(N, Vstack, Icpu1, C, t,V0);
Vplot1(2,:) = V0;
V0 = Series_Model(N, Vstack, Icpu2, C, t,V0);
Vplot1(3,:) = V0;
V0 = Series_Model(N, Vstack, Icpu3, C, t,V0);
Vplot1(4,:) = V0;

subplot(3,2,1), plot(Vplot1);

[p1,pt1] = E2E_Power(Icpu1, Vstack);
[p2,pt2] = E2E_Power(Icpu2, Vstack);
[p3,pt3] = E2E_Power(Icpu3, Vstack);
iter = 28;

ILoad = [.9 .8 1.2 1.1];
V0 = [1,1,1,1];
Iplot = zeros(iter+1, N);
Icheck = zeros(1, iter+1);
taxis = zeros(1, iter+1);
Iplot(1,:) = V0;
Icheck(1) = sum(V0);
for i = 1:iter
    Iplot(i+1,:) = Series_Model(N, Vstack, ILoad, C, t*i, V0, 'current');
    Icheck(i+1) = sum(Iplot(i+1,:));
    taxis(i+1) = i*t;
end
figure(1)
set(gca,'FontSize',14, 'FontName','Times New Roman')
set(gcf,'Color','w')
subplot(3,1,1), plot(taxis,Iplot,'LineWidth',1.5);
set(gca,'FontSize',14, 'FontName','Times New Roman')
title('Current Model')
xlabel('time')
ylabel('Node Voltage (V)')
hleg1 = legend('Node 1','Node 2', 'Node 3', 'Node 4');
set(hleg1,'Location','EastOutside');

RLoad = 1./[.9 .8 1.2 1.1];
V0 = [1,1,1,1];
Rplot = zeros(iter+1, N);
Rcheck = zeros(1, iter+1);
Rplot(1,:) = V0;
Rcheck(1) = sum(V0);
for i = 1:iter
    Rplot(i+1,:) = Series_Model(N, Vstack, RLoad, C, t*i, V0, 'resistance');
    Rcheck(i+1) = sum(Rplot(i+1,:));
end

subplot(3,1,2), plot(taxis,Rplot,'LineWidth',1.5);
set(gca,'FontSize',14, 'FontName','Times New Roman')
title('Resistor Model')
xlabel('time')
ylabel('Node Voltage (V)')
hleg2 = legend('Node 1','Node 2', 'Node 3', 'Node 4');
set(hleg2,'Location','EastOutside');

%subplot(3,2,4), plot(taxis,Rcheck);

PLoad = [.9 .8 1.2 1.1];
V0 = [1,1,1,1];
Pplot = zeros(iter+1, N);
Pcheck = zeros(1, iter+1);
Pplot(1,:) = V0;
Pcheck(1) = sum(V0);
for i = 1:iter
    Pplot(i+1,:) = Series_Model(N, Vstack, PLoad, C, t, Pplot(i,:),
        'dcdc');
Pcheck(i+1) = sum(Pplot(i+1,:));
end
 subplot(3,1,3), plot(taxis,Pplot,'LineWidth',1.5);
set(gca,'FontSize',14, 'FontName','Times New Roman')
title('Power Model')
xlabel('time')
ylabel('Node Voltage (V)')
hleg3 = legend('Node 1','Node 2', 'Node 3', 'Node 4');
set(hleg3,'Location','EastOutside');

%subplot(3,2,6), plot(taxis,Pcheck);

figure(2)
set(gca,'FontSize',14, 'FontName','Times New Roman')
set(gcf,'Color','w')
for i = 1:N
    subplot(N,1,i),
plot(taxis,Iplot(:,i),taxis,Rplot(:,i),taxis,Pplot(:,i),’LineWidth’,1.5);
set(gca,’FontSize’,14, ’FontName’,’Times New Roman’)
top = sprintf(’Node %d’,i);
title(top)
xlabel(’time’)
ylabel(’Node Voltage (V)’)
hleg4 = legend(’Current’,’Resistance’, ’Power’);
set(hleg4,’Location’,’EastOutside’);
end

A.3 PWM and Light Load Operation

This code generates the high resolution PWMs used to control the resonant converter. It also includes light load control based off of the deviation of the node voltages.

/****************************************************************************
! \b External \b Connections 
! Monitor ePWM1-ePWM4 pins on an oscilloscope as described 
! below:
! - ePWM1A is on GPIO0
! - ePWM1B is on GPIO1
! 
! - ePWM2A is on GPIO2
! - ePWM2B is on GPIO3
! 
! - ePWM3A is on GPIO4
! - ePWM3B is on GPIO5
! 
! - ePWM4A is on GPIO6
! - ePWM4B is on GPIO7

#include "DSP2803x_Device.h"  // DSP2803x Headerfile
#include "DSP2803x_Examples.h" // DSP2803x Examples Headerfile
#include "DSP2803x_EPwm_defines.h" // useful defines for initialization
#include "DSP28x_Project.h"  // Device Headerfile and Examples Include File
#include "SFO_V6.h"
// Declare your function prototypes here
//--

void HRPWM1_Config(Uint16);
void HRPWM2_Config(Uint16);
void HRPWM3_Config(Uint16);
void HRPWM4_Config(Uint32);
void HRPWM1_Update(Uint16);
void HRPWM2_Update(Uint16);
void Adc_Config(void);
__interrupt void MainISR(void);
__interrupt void adc_isr(void);

// General System nets - Useful for debug
Uint16 j, DutyFine, n,update;
Uint32 i;

// Global variables used in this example:
Uint16 LoopCount;
Uint16 ConversionCount;
Uint16 Voltage1[10];
Uint16 Voltage2[10];
Uint16 Voltage3[10];
Uint16 Voltage4[10];

Uint32 temp;
int32 V1;
int32 V2;
int32 V3;
int32 V4;
int32 N1;
int32 N2;
int32 N3;
int32 N4;
int32 Vav;
int16 pRes;
int16 pSample;
int16 tol;
int32 llimit;
int32 ulimit;
int32 check;
int32 outbnds;
int32 inbnds;
int32 tav;
int32 t1;
int32 t2;
int32 t3;
int32 t4;

Uint16 PFMflag; // 1 = light load operation, 0 = normal resonant
Uint16 UpdatePeriod;
Uint16 IsrTicker;

void main(void)
{
    PFMflag = 0;
    DutyFine = 0;
    temp=0;
    // *************** Counter Stuff ***********************
    // Step 1. Initialize System Control:
    // PLL, WatchDog, enable Peripheral Clocks
    // This example function is found in the DSP2803x_SysCtrl.c file.
    InitSysCtrl();
    // Step 2. Initialize GPIO:
    // This example function is found in the DSP2803x_Gpio.c file and
// illustrates how to set the GPIO to its default state.
// InitGpio(); // Skipped for this example
// For this case, just init GPIO for EPwm1-EPwm4

// For this case just init GPIO pins for EPwm1, EPwm2, EPwm3, EPwm4
// These functions are in the DSP2803x_EPwm.c file
InitEPwm1Gpio();
InitEPwm2Gpio();
InitEPwm3Gpio();
InitEPwm4Gpio();

// Step 3. Clear all interrupts and initialize PIE vector table:
// Disable CPU interrupts
DINT;

// Initialize the PIE control registers to their default state.
// The default state is all PIE interrupts disabled and flags
// are cleared.
// This function is found in the DSP2803x_PieCtrl.c file.
InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
IER = 0x0000;
IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell
// Interrupt
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example. This is useful for debug purposes.
// The shell ISR routines are found in DSP2803x_DefaultIsr.c.
// This function is found in DSP2803x_PieVect.c.
InitPieVectTable();

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP2803x_InitPeripherals.c
// InitPeripherals(); // Not required for this example
// For this example, only initialize the EPwm
// Step 5. User specific code, enable interrupts:

// counter interrupts

// Reassign ISRs.

EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.EPWM1_INT = &MainISR;
EDIS;

// Enable PIE group 3 interrupt 1 for EPWM1_INT
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;

// Enable CNT_zero interrupt using EPWM1 Time-base
EPwm1Regs.ETSEL.bit.INTEN = 1; // Enable EPWM1INT generation
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO; // Enable interrupt
    CNT_zero event
EPwm1Regs.ETPS.bit.INTPRD = 1; // Generate interrupt on the 1st
    event
EPwm1Regs.ETCLR.bit.INT = 1; // Enable more interrupts

// Enable CPU INT3 for EPWM1_INT:
IER |= M_INT3;

// Enable global Interrupts and higher priority real-time debug
    events:
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM

update =1;
DutyFine =0;
outbnds = 0;
inbnds = 0;
EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;
EDIS;

// Some useful Period vs Frequency values
// SYSCLKOUT = 60 MHz
// ---------------------------
// Period Frequency
// 1000 60 kHz
// 800  75 kHz
// 600  100 kHz
// 500  120 kHz
// 250  240 kHz
// 200  300 kHz
// 100  600 kHz
//  50  1.2 Mhz
//  25  2.4 Mhz
//  20  3.0 Mhz
//  12  5.0 MHz
//  10  6.0 MHz
//   9  6.7 MHz
//   8  7.5 MHz
//   7  8.6 MHz
//   6 10.0 MHz
//   5 12.0 MHz

//====================================================================
// ePWM and HRPWM register initialization
//====================================================================
pRes = 260; //250 for res operation
pSample = 1000;

HRPWM1_Config(pRes);  // ePWM1 target, Period = 10
HRPWM2_Config(pRes);  // ePWM2 target, Period = 20
HRPWM3_Config(pSample);  // ePWM3 target, Period = 10
HRPWM4_Config(30000000);  // ePWM4 target, Period = 20

EPwm1Regs.CMPA.half.CMPAHR = 0; // Left shift by 8 to write into MSB bits
EPwm2Regs.CMPA.half.CMPAHR = 100 << 8; // Left shift by 8 to write
into MSB bits

// ************* ADC Stuff *******************

ConversionCount = 0;
// Interrupts that are used in this example are re-mapped to
// ISR functions found within this file.
EALLOW; // This is needed to write to EALLOW protected register
PieVectTable.ADCINT1 = &adc_isr;
EDIS; // This is needed to disable write to EALLOW protected

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP2803x_InitPeripherals.c
// InitPeripherals(); // Not required for this example
InitAdc(); // For this example, init the ADC
Adc_Config();

// Step 5. User specific code, enable interrupts:

// Enable ADCINT1 in PIE
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Enable INT 1.1 in the PIE
IER |= M_INT1; // Enable CPU Interrupt 1
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt

DBGM
LoopCount = 0;

// VtHi = 6800;
// VtLo = 5800;
for (i=0;i<100000;i++){}
tol = 750;
// start clocks
EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKS = 1;
EDIS;
while (update == 1) {
    N1 = (Voltage1[0]) * 64.68 - 1610;  // 8/4
    N2 = (Voltage2[0]) * 48.62 - 1246;  // 6/6
    N3 = (Voltage3[0]) * 32.52 - 862;    // 4/4
    N4 = (Voltage4[0]) * 16.21 - 435;   // 2/4
    V1 = N1 - N2;
    V2 = N2 - N3;
    V3 = N3 - N4;
    V4 = N4;
    Vav = N1 / 4;
    llim = Vav - tol;
    ulim = Vav + tol;
    check = (V1 > ulim)||(V1 < llim)||(V2 > ulim)||(V2 < llim)||(V3 > ulim)||(V3 < llim)||(V4 > ulim)||(V4 < llim);
    if (check == 1) {
        outbnds++;
        temp = 0;
    }
    if (check == 0) {
        inbnds++;
    }
    if (((check == 0) && (temp == 0))) {
        temp = 1;
        for (i = 0; i < 1000; i++) {}
void HRPWM1_Config(Uint16 period)
{
    // ePWM1 register configuration with HRPWM
    // ePWM1A toggle low/high with MEP control on Rising edge

    EPwm1Regs.TBCTL.bit.PRDLD = TB_SHADOW; // set Immediate load
    EPwm1Regs.TBPRD = period-1; // PWM frequency =
                              // 1 / period
    EPwm1Regs.CMPA.half.CMPA = period / 2-1; // set duty 50%
    EPwm1Regs.CMPA.half.CMPAHR = (1 << 8); // initialize HRPWM
    EPwm1Regs.CMPB = period / 2; // set duty 50%
    EPwm1Regs.TBPHS.all = 0;
    EPwm1Regs.TBPHS.half.TBPHS = 1; // accounts for delay
    EPwm1Regs.TBCTR = 0;

    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP;
    EPwm1Regs.TBCTL.bit.PHSEN = TB_ENABLE; // EPwm1 is the Master
    EPwm1Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN;
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;
    EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;

    EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
    EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;
    EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;

    EPwm1Regs.AQCTLA.bit.ZRO = AQ_CLEAR; // PWM toggle
    low/high
EPwm1Regs.AQCTLA.bit.CAU = AQ_SET;
EPwm1Regs.AQCTLB.bit.ZRO = AQ_CLEAR;
EPwm1Regs.AQCTLB.bit.CBU = AQ_SET;

EALLOW;
EPwm1Regs.HRCNFG.all = 0x0;
EPwm1Regs.HRCNFG.bit.EDGMODE = HR_REP; // MEP control on Rising edge
EPwm1Regs.HRCNFG.bit.CTLMODE = HR_CMP;
EPwm1Regs.HRCNFG.bit.HRLOAD = HR_CTR_ZERO;
EDIS;
}

void HRPWM2_Config(Uint16 period)
{
// ePWM2 register configuration with HRPWM
// ePWM2A toggle low/high with MEP control on Rising edge

EPwm2Regs.TBCTL.bit.PRDLD = TB_SHADOW; // set Immediate load
EPwm2Regs.TBPRD = period - 1; // PWM frequency = 1 / period
EPwm2Regs.CMPA.half.CMPA = period / 2 - 2; // set duty 50% initially
EPwm2Regs.CMPA.half.CMPAHR = (1 << 8); // initialize HRPWM extension
EPwm2Regs.CMPB = period / 2; // set duty 50% initially
EPwm2Regs.TBPHS.all = 0;
EPwm2Regs.TBCTR = 0;

EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP;
EPwm2Regs.TBCTL.bit.PHSEN = TB_ENABLE; // EPwm2 is the Master
EPwm2Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN;
EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;
EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV1;

EPwm2Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
EPwm2Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;
EPwm2Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
EPwm2Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;

EPwm2Regs.AQCTLA.bit.ZRO = AQ_SET; // PWM toggle high/low - ARS
EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;
EPwm2Regs.AQCTLB.bit.ZRO = AQ_CLEAR;
EPwm2Regs.AQCTLB.bit.CBU = AQ_SET;

EALLOW;
EPwm2Regs.HRCNFG.all = 0x0;
EPwm2Regs.HRCNFG.bit.EDGMODE = HR_REP; // MEP control on Rising edge
EPwm2Regs.HRCNFG.bit.CTLMODE = HR_CMP;
EPwm2Regs.HRCNFG.bit.HRLOAD = HR_CTR_ZERO;

EDIS;

void HRPWM3_Config(Uint16 period)
{
    // ePWM3 register configuration with HRPWM
    // ePWM3A toggle high/low with MEP control on falling edge

    EPwm3Regs.TBCTL.bit.PRDLD = TB_IMMEDIATE; // set Immediate load
    EPwm3Regs.TBPRD = 1000-1; // PWM frequency = 1 / period
    EPwm3Regs.CMPA.half.CMPA = 1000 / 2; // set duty 50% initially
    EPwm3Regs.CMPA.half.CMPAHR = (1 << 8); // initialize HRPWM extension
    EPwm3Regs.TBPHS.all = 0;
    EPwm3Regs.TBCTR = 0;

    EPwm3Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP;
    EPwm3Regs.TBCTL.bit.PHSEN = TB_ENABLE; // EPwm3 is the Master
    EPwm3Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_DISABLE;
}
EPwm3Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;
EPwm3Regs.TBCTL.bit.CLKDIV = TB_DIV1;

EPwm3Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
EPwm3Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;
EPwm3Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
EPwm3Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;

EPwm3Regs.AQCTLA.bit.ZRO = AQ_SET; // PWM toggle
    high/low
EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR;
EPwm3Regs.AQCTLB.bit.ZRO = AQ_SET;
EPwm3Regs.AQCTLB.bit.CBU = AQ_CLEAR;

EALLOW;
EPwm3Regs.HRCNFG.all = 0x0;
EPwm3Regs.HRCNFG.bit.EDGMODE = HR_FEP;       // MEP control on
    falling edge
EPwm3Regs.HRCNFG.bit.CTLMODE = HR_CMP;
EDIS;
}

void HRPWM4_Config(Uint32 period)
{
    // ePWM4 register configuration with HRPWM
    // ePWM4A toggle high/low with MEP control on falling edge

    EPwm4Regs.TBCTL.bit.PRDLD = TB_IMMEDIATE;  // set Immediate load
    EPwm4Regs.TBPRD = period-1;                 // PWM frequency =
        1 / period
    EPwm4Regs.CMPA.half.CMPA = period / 2;     // set duty 50%
        initially
    EPwm4Regs.CMPA.half.CMPAHR = (1 << 8);     // initialize HRPWM
        extension
    EPwm4Regs.CMPB = period / 2;               // set duty 50%
        initially
    EPwm4Regs.TBPHS.all = 0;

    57
EPwm4Regs.TBCTR = 0;

EPwm4Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP;
EPwm4Regs.TBCTL.bit.PHSEN = TB_DISABLE;  // EPwm4 is the Master
EPwm4Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_DISABLE;
EPwm4Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;
EPwm4Regs.TBCTL.bitCLKDIV = TB_DIV1;

EPwm4Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
EPwm4Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;
EPwm4Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
EPwm4Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;

EPwm4Regs.AQCTLA.bit.ZRO = AQ_SET;  // PWM toggle high/low
EPwm4Regs.AQCTLA.bit.CAU = AQ_CLEAR;
EPwm4Regs.AQCTLB.bit.ZRO = AQ_SET;
EPwm4Regs.AQCTLB.bit.CBU = AQ_CLEAR;

EALLOW;
EPwm4Regs.HRCNFG.all = 0x0;
EPwm4Regs.HRCNFG.bit.EDGMODE = HR_FEP;  // MEP control on falling edge
EPwm4Regs.HRCNFG.bit.CTLMODE = HR_CMP;
EPwm4Regs.HRCNFG.bit.HRLOAD = HR_CTR_ZERO;
EDIS;
}

void HRPWM1_Update(Uint16 period)
{
// ePWM1 register configuration with HRPWM
// ePWM1A toggle low/high with MEP control on Rising edge

EPwm1Regs.TBCTL.bit.PRDLD = TB_IMMEDIATE;  // set Immediate load
EPwm1Regs.TBPRD = period-1;  // PWM frequency = 1 / period
EPwm1Regs.CMPA.half.CMPA = period / 2-1;  // set duty 50%
void HRPWM2_Update(Uint16 period)
{
// ePWM1 register configuration with HRPWM
// ePWM1A toggle low/high with MEP control on Rising edge

EPwm2Regs.TBCTL.bit.PRDLD = TB_IMMEDIATE; // set Immediate load
EPwm2Regs.TBPRD = period-1; // PWM frequency = 1 / period
EPwm2Regs.CMPA.half.CMPA = period / 2-1; // set duty 50%
initially
EPwm2Regs.CMPA.half.CMPAHR = (1 << 8); // initialize HRPWM extension
EPwm2Regs.CMPB = period / 2; // set duty 50%
initially
EPwm2Regs.TBPHS.all = 0;
EPwm2Regs.TBCTR = 0;
}

void Adc_Config()
{
EALLOW;
AdcRegs.ADCCTL2.bit.ADCNONOVERLAP = 1; // Enable non-overlap mode. This will eliminate 1st sample issue and improve INL/DNL performance.
AdcRegs.ADCCTL1.bit.INTPULSEPOS = 1; // ADCINT1 trips 1 cycle prior to ADC result latching into its result register
AdcRegs.INTSEL1N2.bit.INT1E = 1; // Enabled ADCINT1
AdcRegs.INTSEL1N2.bit.INT1CONT = 0; // Disable ADCINT1
Continuous mode

59
AdcRegs.INTSEL1N2.bit.INT1SEL = 3; // setup EOC5 to trigger ADCINT1 to fire. SEE below two lines. This is why EOC5 sets ADCINT1- it corresponds to the last conversion

AdcRegs.ADCSOC0CTL.bit.CHSEL = 0; // set SOC0 channel select to ADCINB0; SOCx can be set to any ADCINyz
AdcRegs.ADCSOC0CTL.bit.CHSEL = 1; // set SOC1 channel select to ADCINB1
AdcRegs.ADCSOC2CTL.bit.CHSEL = 2; // set SOC2 channel select to ADCINB2
AdcRegs.ADCSOC3CTL.bit.CHSEL = 3; // set SOC3 channel select to ADCINB3

AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 9; // set SOC0 start trigger on EPWM3A
AdcRegs.ADCSOC1CTL.bit.TRIGSEL = 9; // set SOC1 start trigger on EPWM3A, due to round-robin SOC0 converts first then SOC1
AdcRegs.ADCSOC2CTL.bit.TRIGSEL = 9; // set SOC2 start trigger on EPWM3A, due to round-robin SOC1 converts first then SOC2
AdcRegs.ADCSOC3CTL.bit.TRIGSEL = 9; // set SOC3 start trigger on EPWM3A, due to round-robin SOC2 converts first then SOC3

AdcRegs.ADCSOC0CTL.bit.ACQPS = 20; // set SOC0 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
AdcRegs.ADCSOC1CTL.bit.ACQPS = 20; // set SOC1 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
AdcRegs.ADCSOC2CTL.bit.ACQPS = 20; // set SOC2 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
AdcRegs.ADCSOC3CTL.bit.ACQPS = 20; // set SOC3 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
EDIS;

// Assumes ePWM1 clock is already enabled in InitSysCtrl();
EPwm3Regs.ETSEL.bit.SOCAEN = 1; // Enable SOC on A group
EPwm3Regs.ETSEL.bit.SOCASEL = 4; // Select SOC from CPMA on upcount
EPwm3Regs.ETPS.bit.SOCAPRD = 1; // Generate pulse on 1st event
EPwm3Regs.CMPA.half.CMPA = 1500; // Set compare A value
EPwm3Regs.TBPRD = 3000; // Set period for ePWM1
EPwm3Regs.TBCTL.bit.CTRMODE = 0; // count up and start
}

__interrupt void adc_isr(void){
Voltage4[ConversionCount] = AdcResult.ADCRESULT0;
Voltage3[ConversionCount] = AdcResult.ADCRESULT1;
Voltage2[ConversionCount] = AdcResult.ADCRESULT2;
Voltage1[ConversionCount] = AdcResult.ADCRESULT3;
AdcRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //Clear ADCINT1 flag
  reinitialize for next SOC
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Acknowledge interrupt to PIE
return;
}

//MainISR - interrupts at ePWM1 TBCTR = 0.
// This ISR updates the compare and period registers for all ePWM modules within the same period.
// User must ensure that the PWM period is large enough to execute all of the code in the ISR before TBCTR = Period for all ePWM’s.

__interrupt void MainISR(void){
  // Sweep frequency coarsely
if(UpdatePeriod==1){
EPwm1Regs.TBPRD = pRes-1;
EPwm1Regs.CMPA.half.CMPA = pRes/2-1; // set duty 50%

EPwm2Regs.TBPRD = pRes-1;
EPwm2Regs.CMPA.half.CMPA = pRes/2-2; // set duty 50%

UpdatePeriod=0;
}
IsrTicker++;
// Enable more interrupts from this EPWM
EPwm1Regs.ETCLR.bit.INT = 1;
// Acknowledge interrupt to receive more interrupts from PIE group 3
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;

A.4 PWM Sweeping

This code generates the high resolution PWMs used to control the resonant converter. It also sweeps the frequency through the frequency list included. Only the main code is included, it uses the same function calls as the previous code.

#include "DSP2803x_Device.h" // DSP2803x Headerfile
#include "DSP2803x_Examples.h" // DSP2803x Examples Headerfile
#include "DSP2803x_EPwm_defines.h" // useful defines for initialization
#include "DSP28x_Project.h" // Device Headerfile and Examples Include File
#include "SFO_V6.h"
// Declare your function prototypes here

_________________________________________________________________________________

A.4 PWM Sweeping

This code generates the high resolution PWMs used to control the resonant converter. It also sweeps the frequency through the frequency list included. Only the main code is included, it uses the same function calls as the previous code.

#include "DSP2803x_Device.h" // DSP2803x Headerfile
#include "DSP2803x_Examples.h" // DSP2803x Examples Headerfile
#include "DSP2803x_EPwm_defines.h" // useful defines for initialization
#include "DSP28x_Project.h" // Device Headerfile and Examples Include File
#include "SFO_V6.h"
// Declare your function prototypes here

_________________________________________________________________________________
void HRPWM1_Config(Uint16);
void HRPWM2_Config(Uint16);
void HRPWM3_Config(Uint16);
void HRPWM4_Config(Uint32);
void HRPWM1_Update(Uint16);
void HRPWM2_Update(Uint16);
void Adc_Config(void);
__interrupt void MainISR(void);
__interrupt void adc_isr(void);

// General System nets - Useful for debug
Uint16 j, DutyFine, n, update;
Uint32 i;

// Global variables used in this example:
Uint16 LoopCount;
Uint16 ConversionCount;
Uint16 Voltage1[10];
Uint16 Voltage2[10];
Uint16 Voltage3[10];
Uint16 Voltage4[10];

Uint32 flist[16];
int32 findex;

Uint32 temp;
int32 V1;
int32 V2;
int32 V3;
int32 V4;
int32 N1;
int32 N2;
int32 N3;
int32 N4;
int32 Vav;
int16 pRes;
int16 pSample;
int16 tol;
int32 llimit;
int32 ulimit;
int32 check;
int32 outbnds;
int32 inbnds;
int32 tav;

int32 t1;
int32 t2;
int32 t3;
int32 t4;

Uint16 PFMflag;  // 1 = light load operation, 0 = normal resonant
Uint16 UpdatePeriod;
Uint16 IsrTicker;

void main(void)
{
    PFMflag = 0;
    DutyFine = 0;
    temp=0;
    findex=0;
    // **************** Counter Stuff ***********************
    // Step 1. Initialize System Control:
    // PLL, WatchDog, enable Peripheral Clocks
    // This example function is found in the DSP2803x_SysCtrl.c file.
    InitSysCtrl();

    // Step 2. Initialize GPIO:
    // This example function is found in the DSP2803x_Gpio.c file and
    // illustrates how to set the GPIO to its default state.
    // InitGpio(); // Skipped for this example
    // For this case, just init GPIO for EPwm1-EPwm4
// For this case just init GPIO pins for EPwm1, EPwm2, EPwm3, EPwm4
// These functions are in the DSP2803x_EPwm.c file
InitEPwm1Gpio();
InitEPwm2Gpio();
InitEPwm3Gpio();
InitEPwm4Gpio();

// Step 3. Clear all interrupts and initialize PIE vector table:
// Disable CPU interrupts
DINT;

// Initialize the PIE control registers to their default state.
// The default state is all PIE interrupts disabled and flags
// are cleared.
// This function is found in the DSP2803x_PieCtrl.c file.
InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
IER = 0x0000;
IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example. This is useful for debug purposes.
// The shell ISR routines are found in DSP2803x_DefaultIsr.c.
// This function is found in DSP2803x_PieVect.c.
InitPieVectTable();

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP2803x_InitPeripherals.c
// InitPeripherals(); // Not required for this example

// For this example, only initialize the EPwm
// Step 5. User specific code, enable interrupts:
// counter interrupts

// Reassign ISRs.

EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.EPWM1_INT = &MainISR;
EDIS;

// Enable PIE group 3 interrupt 1 for EPWM1_INT
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;

// Enable CNT_zero interrupt using EPWM1 Time-base
EPwm1Regs.ETSEL.bit.INTEN = 1; // Enable EPWM1INT generation
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO; // Enable interrupt
  CNT_zero event
EPwm1Regs.ETPS.bit.INTPRD = 1; // Generate interrupt on the 1st
  event
EPwm1Regs.ETCLR.bit.INT = 1; // Enable more interrupts

// Enable CPU INT3 for EPWM1_INT:
IER |= M_INT3;

// Enable global Interrupts and higher priority real-time debug
  events:
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM

update =1;
DutyFine =0;
outbnds = 0;
inbnds = 0;
EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKSRC = 0;
EDIS;

// Some useful Period vs Frequency values
SYSCLKOUT = 60 MHz

---

<table>
<thead>
<tr>
<th>Period</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>60 kHz</td>
</tr>
<tr>
<td>800</td>
<td>75 kHz</td>
</tr>
<tr>
<td>600</td>
<td>100 kHz</td>
</tr>
<tr>
<td>500</td>
<td>120 kHz</td>
</tr>
<tr>
<td>250</td>
<td>240 kHz</td>
</tr>
<tr>
<td>200</td>
<td>300 kHz</td>
</tr>
<tr>
<td>100</td>
<td>600 kHz</td>
</tr>
<tr>
<td>50</td>
<td>1.2 MHz</td>
</tr>
<tr>
<td>25</td>
<td>2.4 MHz</td>
</tr>
<tr>
<td>20</td>
<td>3.0 MHz</td>
</tr>
<tr>
<td>12</td>
<td>5.0 MHz</td>
</tr>
<tr>
<td>10</td>
<td>6.0 MHz</td>
</tr>
<tr>
<td>9</td>
<td>6.7 MHz</td>
</tr>
<tr>
<td>8</td>
<td>7.5 MHz</td>
</tr>
<tr>
<td>7</td>
<td>8.6 MHz</td>
</tr>
<tr>
<td>6</td>
<td>10.0 MHz</td>
</tr>
<tr>
<td>5</td>
<td>12.0 MHz</td>
</tr>
</tbody>
</table>

---

ePWM and HRPWM register initialization

---

Res = 200; // 250 for res operation
Resample = 1000;

HRPWM1_Config(Res); // ePWM1 target, Period = 10
HRPWM2_Config(Res); // ePWM2 target, Period = 20
HRPWM3_Config(Resample); // ePWM3 target, Period = 10
HRPWM4_Config(30000000); // ePWM4 target, Period = 20

EPwm1Regs.CMPA.half.CMPAHR = 0; // Left shift by 8 to write into MSB bits
EPwm2Regs.CMPA.half.CMPAHR = 100 << 8; // Left shift by 8 to write into MSB bits
ConversionCount = 0;
// Interrupts that are used in this example are re-mapped to
// ISR functions found within this file.
EALLOW; // This is needed to write to EALLOW protected register
PieVectTable.ADCINT1 = &adc_isr;
EDIS;   // This is needed to disable write to EALLOW protected
// registers

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP2803x_InitPeripherals.c
// InitPeripherals(); // Not required for this example
InitAdc(); // For this example, init the ADC
Adc_Config();

// Step 5. User specific code, enable interrupts:

// Enable ADCINT1 in PIE
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Enable INT 1.1 in the PIE
IER |= M_INT1; // Enable CPU Interrupt 1
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt
        DBGM

LoopCount = 0;

// VtHi = 6800;
// VtLo = 5800;
for (i=0;i<100000;i++){}
tol = 750;
// start clocks
EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
EDIS;

flist[0] = 50000;
flist[1] = 100000;
flist[2] = 150000;
flist[3] = 200000;
flist[4] = 250000;
flist[5] = 300000;
flist[6] = 400000;
flist[7] = 500000;
flist[8] = 600000;
flist[9] = 800000;
flist[10] = 1000000;
flist[12] = 1400000;
flist[13] = 1600000;
flist[14] = 1800000;
flist[15] = 2000000;
UpdatePeriod =0;

while (update ==1){
    // while (UpdatePeriod == 1){}
    //ensure that period is even
    pRes = 30000000/flist[findex];
    pRes = pRes*2;
    UpdatePeriod = 1;

    for(i=0;i<100000;i++){
        for(j=0;j<300;j++){}
    }
    findex = findex+1;
    if (findex == 12){
        findex = 0;
    }
}
}
REFERENCES


