Distributed, Modular, Open Control Architecture for Power Conversion Systems

Jinghong Guo

Dissertation submitted to Faculty of Virginia Polytechnic Institute and State University in fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

May 13, 2005

Blacksburg, Virginia

Keywords: dataflow, open control architecture, PEBB, real-time system

DISTRIBUTED, MODULAR, OPEN CONTROL ARCHITECTURE FOR POWER CONVERSION SYSTEMS

Jinghong Guo

(Abstract)

Due to close coupling to hardware and lack of software engineering technologies, the control software in digitally controlled power conversion systems is difficult to design and maintain. This is a natural consequence of a topology- or application-driven design approach. This research work proposes a distributed, modular, open control architecture for power conversion systems to reduce control design complexity, encapsulate and localize design dependencies, reduce unnecessary redesign effort and improve software quality. Dataflow style is chosen as the architectural style for the proposed control architecture based on comparative analysis. The detailed implementation of the dataflow architecture is presented. The resulting dataflow control software is evaluated in comparison to the legacy approach to control design used in industry and academia. The dataflow control software for a 3-phase voltage source inverter is also tested on a real PEBB-based converter system. To further explore the flexibility of control composition that is brought by the dataflow approach, the feasibility of dynamic control reconfiguration is also presented as an important future research direction.

To my parents, Longsheng Guo and Yu Wang

ACKNOWLEDGEMENTS

When I fist came to the Center for Power Electronics Systems (CPES) at Virginia Tech, I barely knew a single terminology in power electronics. Finally I come along all the way. I would never achieve this without all the help and support from so many people.

First I would like to acknowledge my advisors Dr. Dushan Boroyevich and Dr. Stephen Edwards for their invaluable guidance, support and encouragement that carried me through this research work. They are not only advisors to me, but also taught me the paths to successful research and leadership.

I am also grateful for my committee members Dr. Fred Wang, Dr. Binoy Ravindran and Dr. Douglas Lindner for their deep insight to different research areas other than power electronics and challenging discussions.

I would like to thank my colleagues in CPES, Dr. Rolando Burgos, Mr. Jerry Francis, Mr. Daniel Ghizoni, Mr. Sebastian Rosado, Dr. Kun Xing, Mr. Dengming Peng, Mr. Wei Shen, Ms. Qian Liu, Mr. Ivan Celanovic, Mr. Tim Thacker, for their cooperation, support and valuable discussions, which made my research work at CPES more productive.

CPES has such a wonderful staff and administrative team, Ms. Trish Rose, Ms. Marianne Hawthorne, Ms. Teresa Shaw, Ms. Elizabeth Tranter, Mr. Robert Martin, Ms. Michelle Czamanske, Ms. Ann Craig and Mr. Jamie Evans. I wish to thank them for their support and hard work.

I wish to thank Dr. Ming Xu, Dr. Yuancheng Ren for helping me on the study of power management circuits.

Special thanks go to all my friends in Blacksburg, Ms. Yan Jiang, Ms. Chunchun Xu, Mr. Chuanyun Wang, Dr. Qun Zhao, Dr. Bo Yang, Dr. Linyin Zhao, Dr. Gary Yao, etc., for all the wonderful time we spent together.

Finally, I would dedicate my sincere gratitude to my parents and brother. For all these years, it is their selfless love and inspirations giving me the strength to defeat confusions, frustrations and difficulties along the journey. I am not afraid because I know they are always by my side.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

1.1 Scope of Research Interest: Control Design for Medium- to High-Power Conversion Systems

In power conversion systems, analog controllers, based on opamps and comparators or pre-calculated PROM-based look-up tables, are widely used for applications such as low-power DC-DC converters that encompass relatively simple control algorithms but stringent requirements for high switching frequency [1]-[7]. But for more complex converters used for medium- to high-power conversion applications, such as three-phase converters, power factor correction converters, matrix converters, etc., a digital controller is a more suitable choice due to more complicated control algorithms and the availability of fast and inexpensive digital components [8]-[13].

This research work will concentrate on the digital control design and implementation for medium- to high-power conversion systems. Figure 1-1 shows the general system structure of a digitally controlled power converter [14]-[16]. The analog feedback information from the power stage is acquired via sensors, and is then converted to a digital format through analog-to-digital converters (ADCs). The digital controller manipulates the feedback information from the power stage and generates digital control commands for the switching network, which in turn are translated to analog gate signals through digital-to-analog converters (DACs).

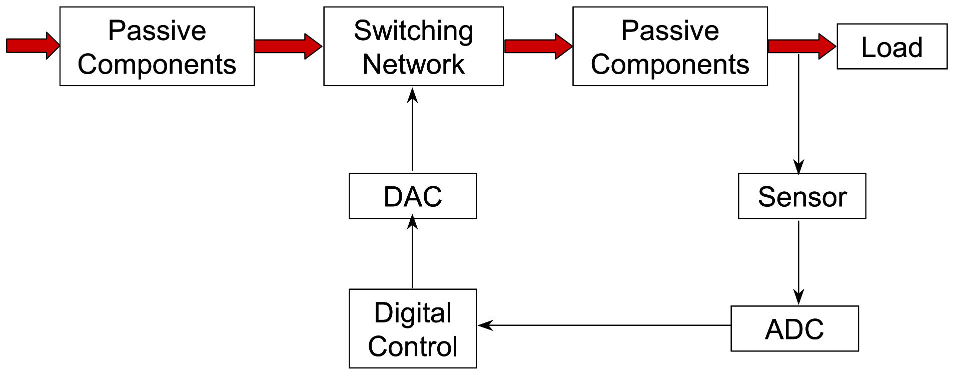


Figure 1-1. Digitally controlled power converter system.

Digital control of power converters has the following basic characteristics:

- It is embedded: The controller is hardware-oriented and has restricted resources [17]; and
- it is hard real-time: control commands have to be updated for each switching period [18].

Until quite recently, the legacy approach to the study, analysis and design of static power converters were still topology- or application-oriented. The actual production process of converters still suffered from a complete lack of generality in its approach [19]-[21], the enhancement of which requires an enormous engineering effort to develop individual converters, thus detrimentally affecting both production costs and revenues for the power electronics industry. Some research has focused on this situation, actually spawning the power electronics building block (PEBB) concept [22]-[28], in which conventional power electronics tasks are performed by intelligent power processing modules capable of interconnecting with their peers. This concept attempts to provide the means to greatly simplify, standardize and generalize the analysis, design, production and usage of static power converters. However, similar situations in the control design for power converters have not gained sufficient attention, but will inevitably hinder the modularization and standardization process in power conversion systems.

1.2 The Problem: Closely Integrated Control Software with System Hardware

Because control software is easier to modify compared to other design domains, it is normally left as the last area to be implemented in the whole system design. The legacy approaches to the control development of power conversion systems, mainly in monolithic assembly or C language, is also per-application-based, and the dependencies of control software on other design areas of the system are interwoven [14]-[16], [29]-[34]. Very often the piece of software that conducts the control algorithm is also tangled with code served as interface to both control host hardware and controlled devices. Due to both the close coupling to hardware and the lack of application of software engineering technologies, the control software in the digitally controlled power conversion system is difficult to design and maintain, which in turn causes the following problems:

1. Multi-disciplinary expertise is required for the individual control software designer: A single control designer needs to know every detail of control theory, control host hardware, controlled devices, and even digital communication techniques. The control construction is time consuming and error prone;
2. Software development is highly dependent on hardware; this dependency and the lack of well-defined application programming interfaces (APIs) complicate system integration and maintenance;
3. Quality achieved in prior software designs does not carry over well;
4. New functionality or components are hard to add to existing systems; and
5. The low adaptability to changes in the application requirements and runtime environment leads to a tremendous amount of redesign effort needed for each new system.

1.3 Research Objectives: Decouple Control Software from Hardware

The research goal of this work is to explore the possibilities for designing a distributed, modular, open control architecture for power conversion systems, and to evaluate its advantages and disadvantages. The control architecture presented in this work should achieve the following goals.

1. **Reduce coupling in order to:**
   - ➢ Make converter control applications independent of control host hardware;
   - ➢ Make converter control algorithms independent of controlled devices; and
   - ➢ Make converter control applications easily adapted to the distributed operating environment.
2. **Separate software structure from control functionality in order to:**
   - ➢ Make commonly used functionality to be reused in diverse control applications;

- Make control software to be scalable to meet specific application requirements; and
- Make new functionality to be easily integrated into existing systems.

1.4 Approach: Distributed, Modular, Open Control Architecture

To address the problems mentioned above, this research work does not emphasize the functionality of power converter control algorithms, but concentrates instead on the design methodology and software engineering aspects that could benefit control design in a wide array of power conversion systems. The drawbacks of legacy software design approaches originate from their monolithic software structure, which achieves customization at computation capacity but at the cost of flexibility. This research work explores the feasibility of applying software engineering techniques to shift the embedded control of power conversion systems from the sole choice of control algorithms and data structures to a well-structured, open software architecture paradigm based on dataflow software style and a scalable lightweight real-time operating system. Since power conversion system control is hard real-time, the open control architecture presented in this work attempts to balance the maximum flexibility with the performance of the control software.

1.5 Research Contributions

The main contributions of this research work are:

1. Gave comparative analysis of current practices for control software design in power conversion systems;
2. Proposed a concept of distributed, modular, open control architecture for power conversion systems;
3. Based on comparative analysis between mainstream software styles, applied dataflow to implement the proposed open control architecture;
4. Experimentally verified and evaluated the dataflow-based open architecture; and
5. Explored the feasibility towards reconfigurable control from the software implementation point of view.

1.6 Dissertation Outline

The rest of this dissertation is organized as follows:

In Chapter 2, the characteristics of control in power conversion systems will be analyzed from both control scheme and real-time embedded points of view.

In Chapter 3, the current practices of control design in digitally controlled power conversion systems will be analyzed to show their major drawbacks and their difficulty in meeting the trends for power conversion system development. Related work is also investigated in this chapter in order to point out the potential methodologies that might solve similar issues encountered in motion control and manufacture communities. The distributed, modular and open control architecture is presented based on the analysis of the interactions between control and other design domains in a power conversion system. This chapter will also show how this open architecture is able to both loosen the coupling between control development and the system hardware platform, and to separate the software structure in order to control functionality.

Chapter 4 will discuss the importance of applying software styles. Based on the analysis of characteristics of power conversion control and the targets of the open control architecture, the requirements of implementation of the proposed open control architecture are pronounced. Comparative analysis shows that dataflow is a strong candidate.

Chapter 5 will show how to map the open control architecture through dataflow software style. The control function building blocks (so called elementary control objects), the dataflow graph that represents the control algorithm and an especially design real-time kernel will be presented in detail.

In Chapter 6, several dataflow-based converter control applications will be presented as design examples to illustrate how to implement control within the proposed open control architecture. The assessments of the dataflow style control software, including feasibility and engineering efforts, are conducted through comparison to equivalent control programs implemented in monolithic C and Real-time Workshop Embedded

Coder. The closed-loop control for a 3-phase voltage source inverter is also tested on a real PEBB-based converter system.

In Chapter 7, as an important future research direction, the feasibility of incorporating online dynamic reconfigurability into the open control architecture is investigated. This chapter will also conclude the whole research work presented in this dissertation.

Chapter 2: CHARACTERISTICS OF CONTROL IN POWER CONVERSION SYSTEMS

Since the research interest of this work is to apply software engineering techniques to design control software for power conversion systems, this chapter will give reviews of main design issues and technology background in the related problem domains.

Because three-phase pulse-width modulation (PWM) converters are widely used in medium- to high- power conversion applications, Section 2.1 gives a review of the major control schemes and techniques used in the these converters. As mentioned in Chapter 1, digital control for power conversion systems is real-time and embedded, which often leads to constraints for control design. Section 2.2 will describe the control from real-time embedded system point of view.

2.1 An Overview of Control Schemes of Power Converters

Three-phase PWM converters are widely used in medium- to high-power conversion applications. We will take the control of those converters as examples to show the diversity of control schemes used in power conversion systems. Three-phase PWM converters can be categorized as either voltage-fed or current-fed converters, according to their DC-side connections, as shown in Figure 2-1 and Figure 2-2. Based on the power-flow direction, these are either rectifiers (power flows from AC to DC) or inverters (DC to AC).

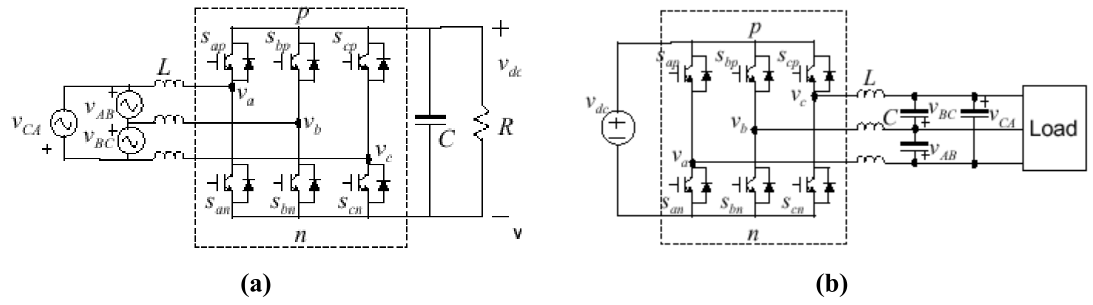
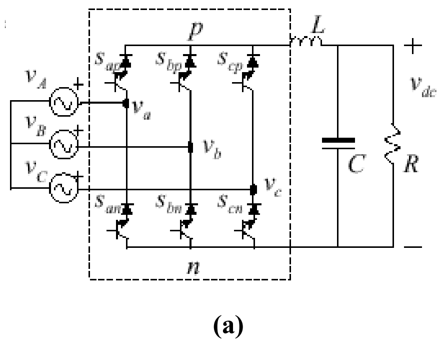


Figure 2-1. Voltage-fed converters: (a) boost rectifier and (b) voltage-source inverter (VSI).



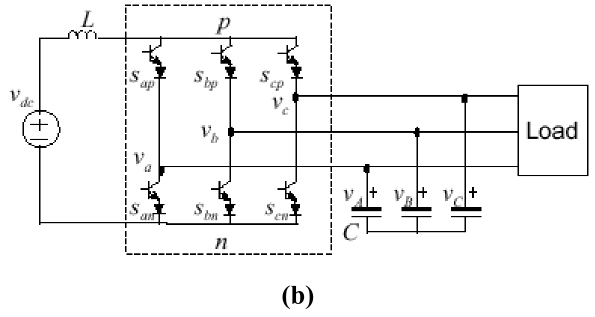


Figure 2-2. Current-fed converters: (a) buck rectifier and (b) current-source inverter (CSI).

Voltage-fed converters are widely used in AC motor drives, AC uninterruptible power supplies (UPS), and AC power supplies from batteries, fuel cells, active harmonic filters, etc. Current-fed converters have such applications as speed control for large power induction and synchronous motors, the variable-frequency starting of 60 Hz wound-field synchronous motors, super-conducting magnet energy storage, DC motor drives, etc [35], [36].

The generic control structure of three-phase PWM converters shown in Figure 2-3 illustrates that the control normally has three components – modulator, current control (or inner loop control) and output control (or outer loop control).

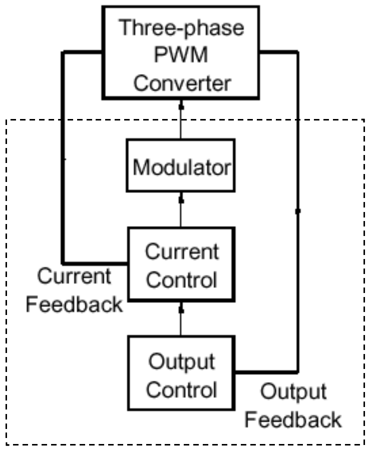


Figure 2-3. Generic control structure of three-phase converters.

2.1.1 Open-Loop Control and Modulation Techniques

An open-loop PWM three-phase converter uses only a modulator. The widely used modulation techniques fall into two categories: carrier-based modulation and space-vector modulation (SVM).

2.1.1.1 Sinusoidal PWM

The well-known triangular carrier-based Sinusoidal PWM (SPWM) for three-phase static converter control was originally proposed by Schonung and Stemmler in 1964 [37]. By comparing the common carrier signal with three reference sinusoidal signals, the logical signals defining the switching instants are generated, as shown in Figure 2-4.

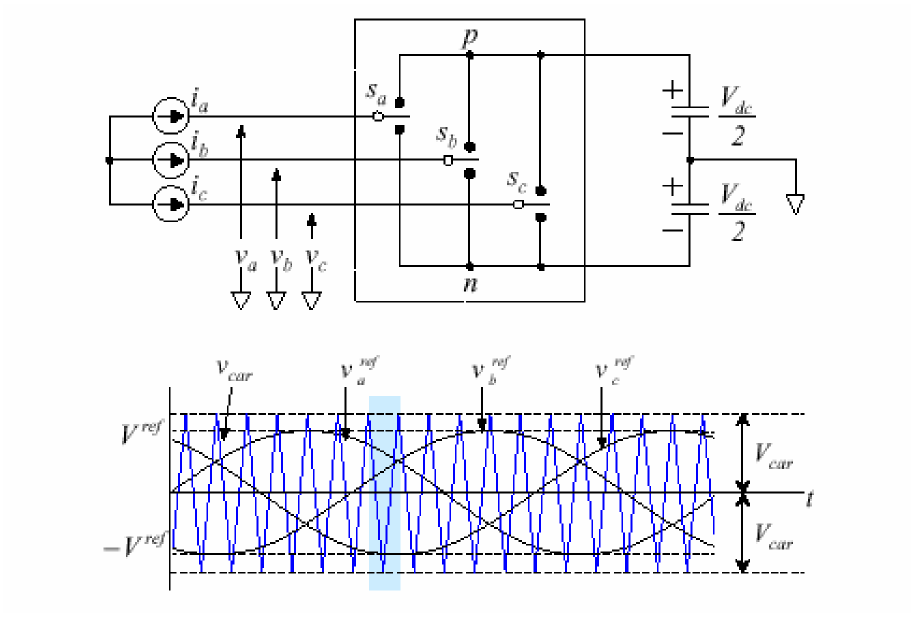
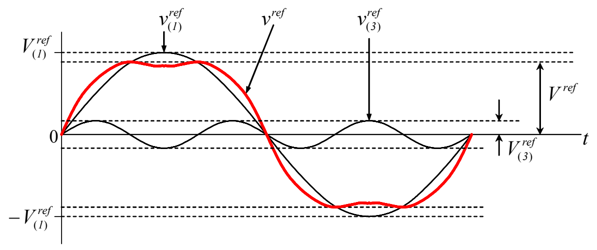


Figure 2-4. SPWM.

2.1.1.2 Carrier-Based PWM with Zero-Sequence Injection

SPWM is relatively simple to implement, but its reduced range of linearity is a major problem. Hybrid or generalized PWM was proposed to extend the linear region of the modulator, thus improving its performance in terms of average switching frequency and current ripple reduction [48]. If the neutral point on the AC side of power converter N is not connected to the DC-side midpoint, one more degree of freedom for design appears as the partitioning of two zero states [49]. Any zero-sequence signal can be injected into the reference modulation waves to widen the linear operation range of modulation without affecting the phase-to-phase voltage at the load side [50]. Several modulation methods have been developed based on this observation. Among them there are the third-harmonic injection PWM (THIPWM) [51] and the discontinuous PWM methods (including DPWM, DPWM1 and DPWM2). Figure 2-5 shows the method of third-harmonic injection. Figure 2-6 shows the general block diagram of carrier-based PWM with zero-sequence injection modulation methods.



$$v^{ref} = V_{(1)}^{ref} \cdot \sin \omega t + V_{(3)}^{ref} \cdot \sin 3\omega t$$

$$V^{ref} = V_{(1)}^{ref} + V_{(3)}^{ref}$$

Figure 2-5. Third-harmonic injection PWM.

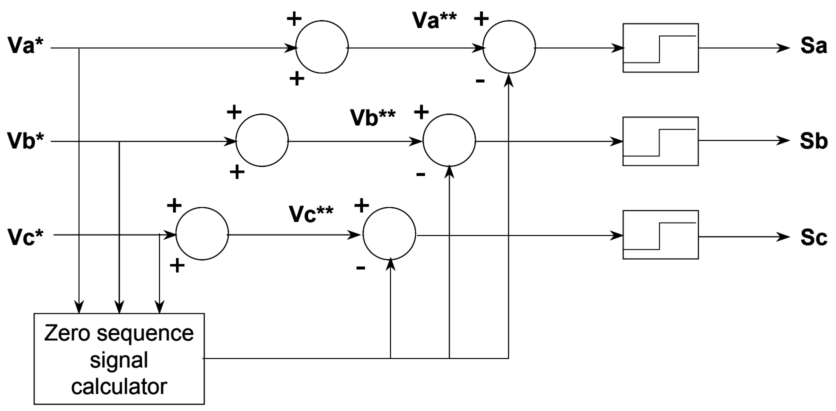


Figure 2-6. Carrier-based PWM with zero-sequence injection modulation.

2.1.1.3 Space-Vector Modulation

The Space-Vector Modulation (SVM) strategy is based on space-vector representation of the converter's AC side voltage or current, and has become very popular because of its simplicity [52], [53]. A three-phase, two-level converter provides eight possible switching states, made up of six active and two zero switching states. Active vectors divide the plane into six sectors, using a reference vector Vref that is synthesized by switching two adjacent vectors on for proper lengths of time. It is possible to implement vector Vref using different switch on/off sequences for V1, V2 and the zero vectors, as shown in Figure 2-7 (a). The general control scheme of SVM is shown in Figure 2-7 (b). In contrast to carrier-based PWM, in the SVM there are no separate modulators for each phase. A reference vector is synthesized by vector combinations made from the eight switching states. By arranging the two zero switching states, different SVM schemes can be generated in order to reduce switching loss or total harmonic distortion (THD) [48], [55]-[60].

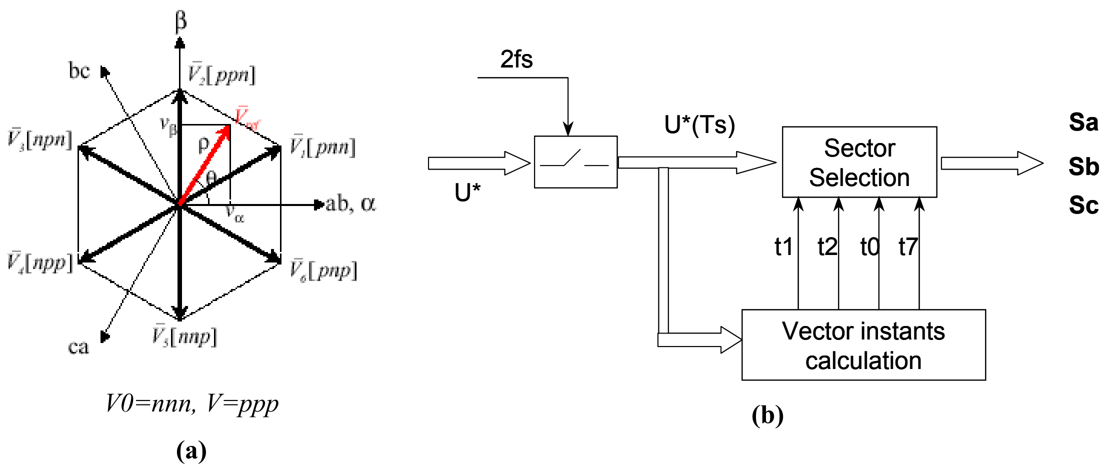


Figure 2-7 . Space vector modulation: (a) vector synthesis and (b) control scheme.

2.1.2 Closed-Loop Control

Most applications of three-phase voltage-source PWM converters—as motor drives, active filters, high-power-factor AC/DC converters, UPS systems, and AC power supplies—have a control structure comprising an internal current feedback loop. Consequently, the performance of the converter system largely depends on the quality of the applied current control strategy. Existing current control techniques can be classified in various ways [61]-[65]. Figure 2-8 shows one way of classification.

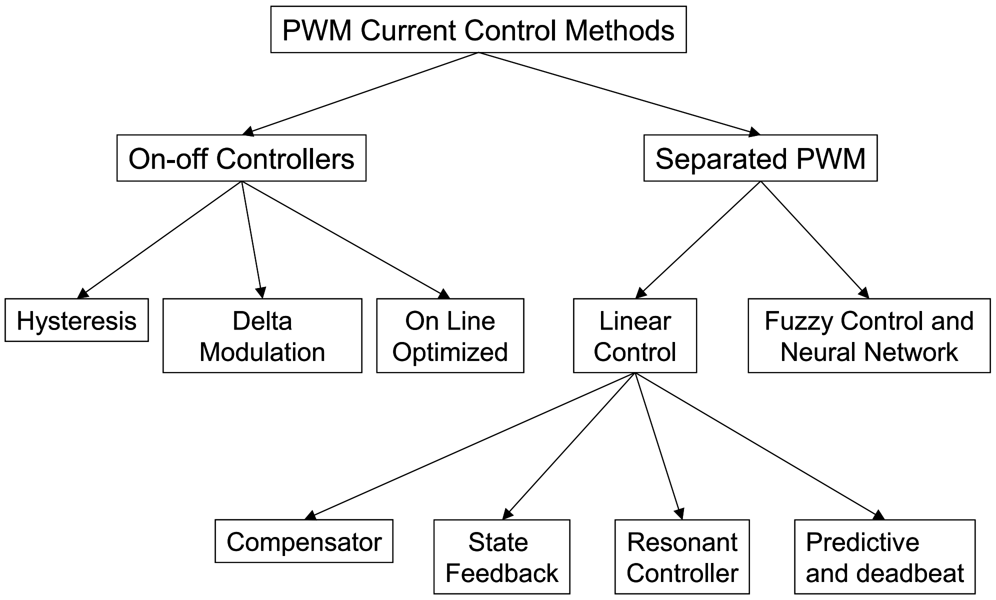


Figure 2-8. Current control techniques.

Most of these control algorithms are explained below.

2.1.2.1 Linear Controller

A. Linear Compensator

A.1 Ramp Comparison Controller

The ramp comparison current controller [66] uses the output-current ripple and feedback to influence the switching instants, as shown in Figure 2-9.

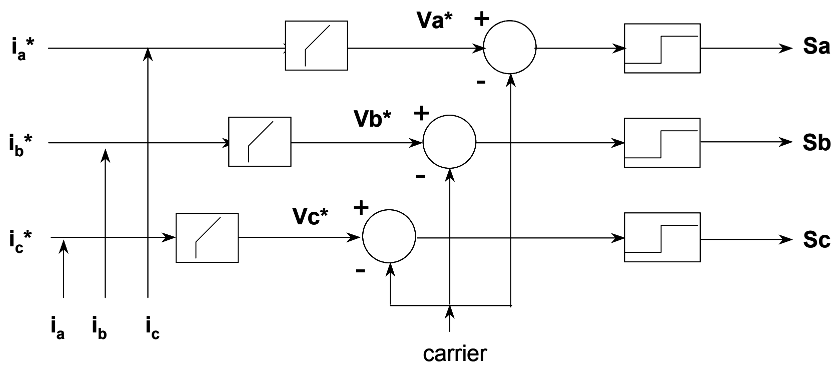


Figure 2-9. Ramp comparison current controller.

A.2 Stationary Vector Controller

In the three-phase isolated neutral-load topology, the three-phase current must add up to zero. Therefore, only two linear compensators are necessary, and the three-phase inverter reference voltage signals can be established algebraically using two-to-three phase conversion blocks αβ/abc, as shown in Figure 2-10.

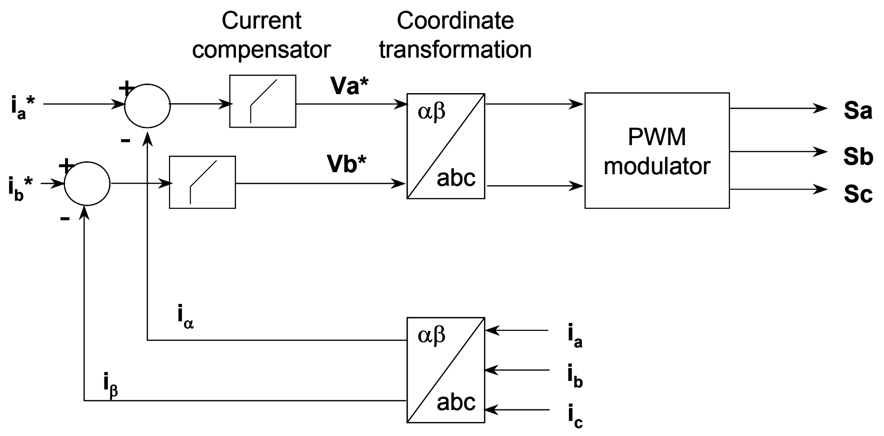


Figure 2-10. Stationary current controller.

A.3 Synchronous Vector Controller [67]-[69]

In many industrial applications, an ideally precise current is required, because even small errors in phase or amplitude cause incorrect system operation. In such cases, the synchronous controller is applied. In this approach, the current vector sits in rotating coordinates dq, which, after the coordinates' transformation, become DC components, as shown in Figure 2-11.

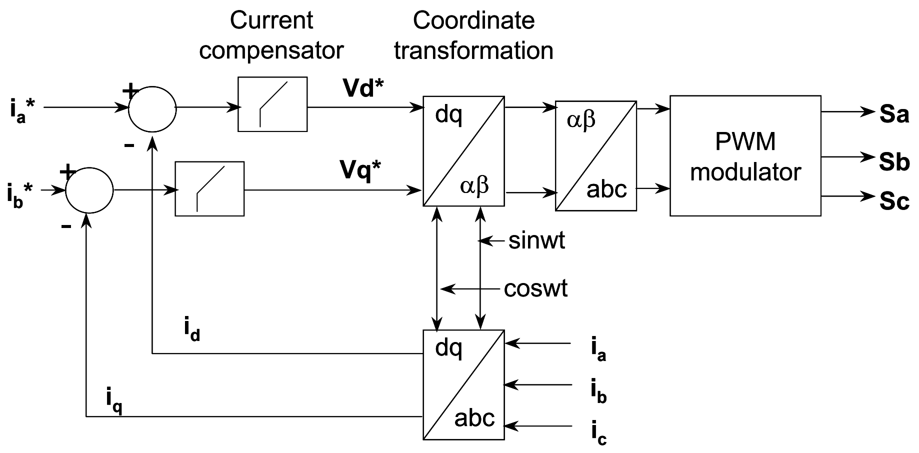


Figure 2-11. Synchronous vector current controller.

B. State-Feedback Current Controller

A state-feedback controller [70]-[72], working in stationary or synchronous rotating coordinates to increase the dynamic response of the system, can replace the conventional PI compensators used in current error compensation. Figure 2-12 shows a state-feedback controller.

C. Constant-Switching-Frequency Predictive Controller

In the case of the constant-switching-frequency predictive controller [74], [75], as shown in Figure 2-13, the algorithm is used to calculate the voltage-vector commands once per sample period Ts, and these commands will force the actual current accordingly. The calculated voltage is then implemented in the sinusoidal or space-vector PWM algorithm. The predictive controller can be implemented in either stationary or synchronous coordinates.

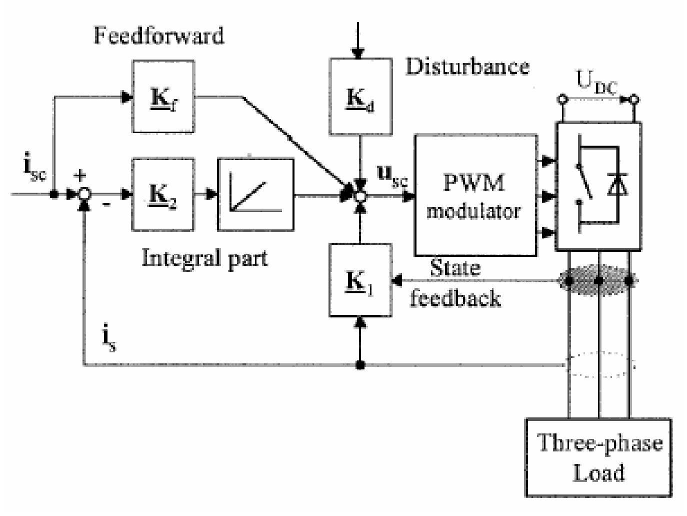


Figure 2-12. State-feedback controller [74].

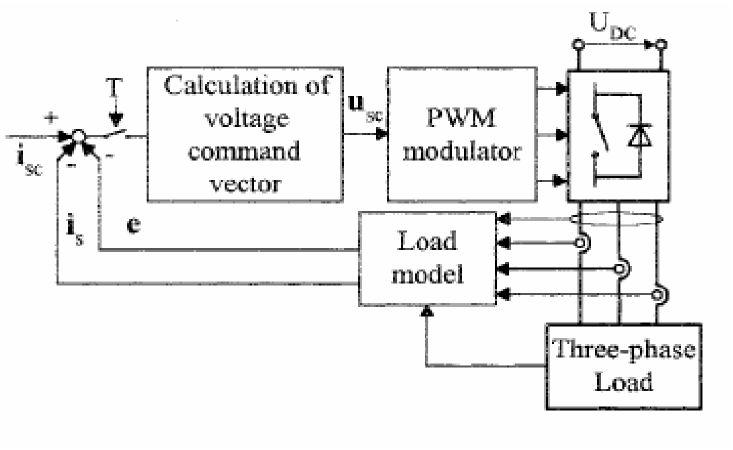


Figure 2-13. Predictive current controller [75].

D. On-Line Optimization

This class of controllers performs a real-time optimization algorithm [75]-[76] and requires complex on-line calculations, which usually can be implemented only on microprocessors, as shown in Figure 2-14.

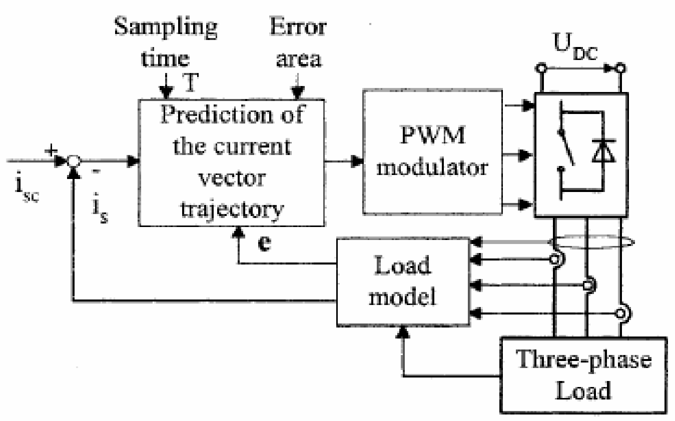


Figure 2-14. On-line optimization controller [76].

E. Delta Modulation

The basic scheme of delta modulation [77] is shown in Figure 2-15. Only the error sign is detected by the comparators, whose outputs are sampled at a fixed rate so that the inverter status is kept constant during each sampling interval.

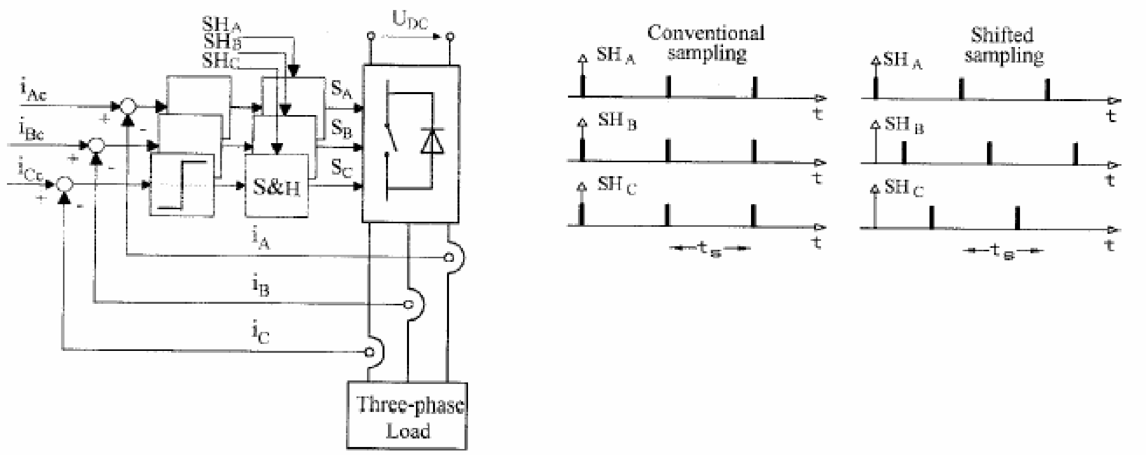


Figure 2-15. Delta-modulation controller [77].

2.1.3 Outer-Loop Control

The outer-loop control is to control output voltage, torque or flux. Numerous control techniques are used in outer-loop control, depending on the application. For example, in induction motor applications, the control methods can be classified as shown in Figure 2-16 [78]-[80]. Figure 2-17 shows a typical field-oriented controller [81].

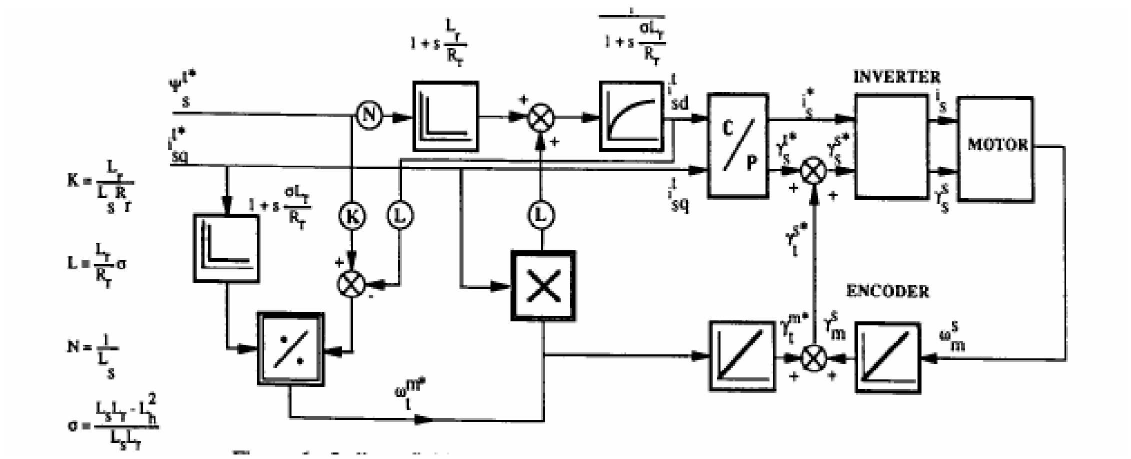

INVERTER
MOTOR
ENCODER
C/P
$K = \frac{L_r}{L_s R_r}$
$L = \frac{L_r}{R_r}\sigma$
$N = \frac{1}{L_s}$
$\sigma = \frac{L_s L_r - L_h^2}{L_s L_r}$

Figure 2-17. Indirect field orientation in the stator flux [81].

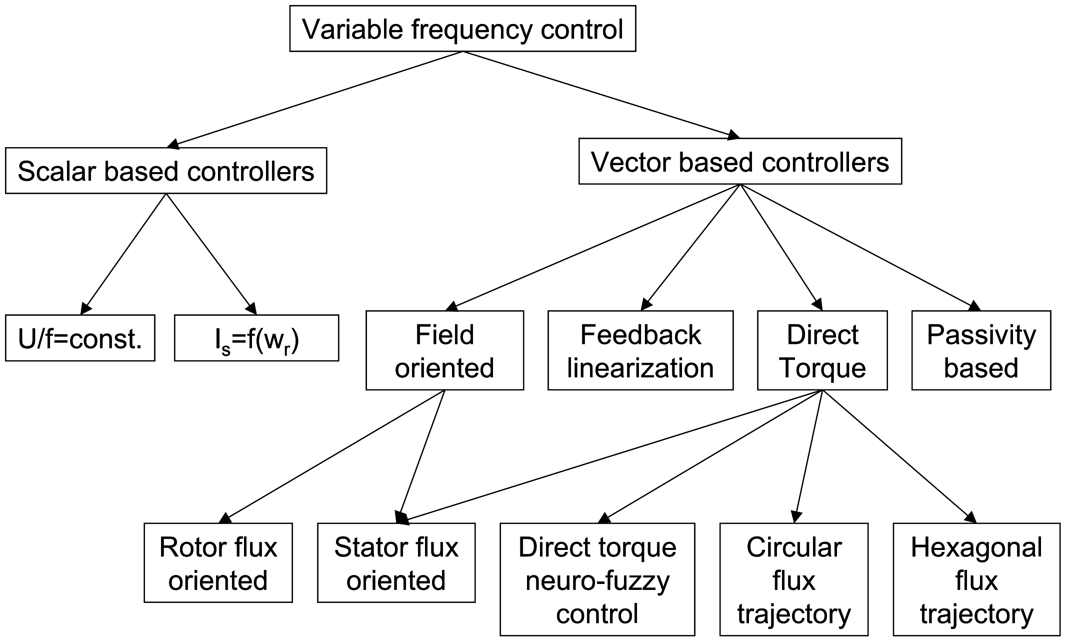
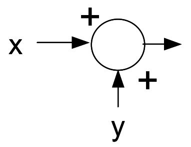
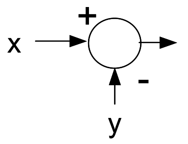
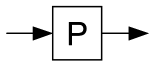
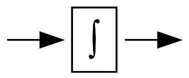
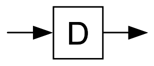
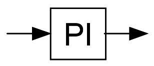
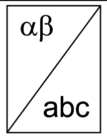


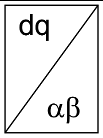
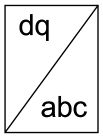
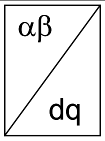
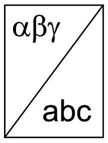
Figure 2-16. Classification of induction motor control methods.

2.1.4 Potential of Modularization of Control in Power Conversion Systems

The comparative analysis of various control schemes shows that a control scheme is composed of some independent function blocks that can also be used in other control schemes; examples of these are comparators, integrators, etc. Table 2-1 lists some math function blocks that are widely used in power converter control.

Table 2-1. A partial list of math function blocks.

Function Name	**Symbol**	**Math Expression**
Single Input/Output		
Algebraic Functions		
Sum-2		$z = x + y$
Sub-2		$z = x - y$
Time-Domain Functions		
Sinwt		$z = \sin(wt)$
Coswt		$z = \cos(wt)$
S-Domain Functions		
Proportional (P)		$G(s) = K_P$
Integrator (I)		$G(s) = \frac{1}{K_I s}$
Derivative (D)		$G(s) = K_D s$
PI		$G(s) = K_P + \frac{1}{K_I s}$
PID		$G(s) = K_P + \frac{1}{K_I s} + K_d s$
Multi-Input/Output		
Algebraic Functions		
abc_to_αβ		$T_{\alpha\beta/abc} = \sqrt{\frac{2}{3}}\begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix}$

αβ_to_dq		$T_{dq/\alpha\beta}=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}$
abc_to_dq		$T_{dq/abc}=\sqrt{\frac{2}{3}}\begin{bmatrix}\cos\theta & \cos(\theta-\frac{2\pi}{3}) & \cos(\theta+\frac{2\pi}{3})\\ -\sin\theta & -\sin(\theta-\frac{2\pi}{3}) & -\sin(\theta+\frac{2\pi}{3})\end{bmatrix}$
dq_to_αβ		$T_{\alpha\beta/dq}=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$
abc_to_αβγ		$T_{\alpha\beta/abc}=\sqrt{\frac{2}{3}}\begin{bmatrix}1 & -1/2 & -1/2\\ 0 & \sqrt{3}/2 & -\sqrt{3}/2\\ 1/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2}\end{bmatrix}$

Digital implementation of the time-domain math functions is straightforward. For s-domain transfer functions, they can be mapped to the z-domain, and can then be implemented digitally. Otherwise, PI controllers can be designed directly in the z-domain to facilitate digital implementation.

There are also some commonly used logic function blocks. The modulation part of power converter control is more logical than mathematical. A typical comparator, as shown in Figure 2-18 (a), normally has two inputs – an error signal and a carrier -- and one two-logic output. When the error value is less than the carrier, the output remains at logic 0; otherwise, the output logic will be 1. The comparator is normally implemented as a timer. The space-vector modulator requires more complex logic implementation, as described in Section 2.1.1.

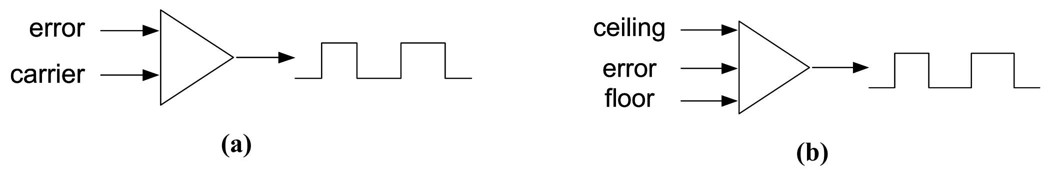


Figure 2-18. Comparators: (a) single comparison and (b) hysteresis comparison.

As a summary, it is obvious that the control algorithm can be composed from commonly used function blocks.

2.2 Real-Time Embedded Control

Generally speaking, a real-time system is one that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure [82]. To be more specific, a real-time computer system is one in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced [83]. Real-time systems are often reactive or embedded systems. Reactive systems are those that have some ongoing interaction with their environment. Embedded systems are those used to control specialized hardware in which the computer system is installed. Based on the above definitions, the control subsystem for power conversion systems can be defined as a real-time embedded control system. The following subsections will present some of the main characteristics of the real-time embedded system and its major design considerations.

2.2.1 Principles

Typically, a real-time embedded system consists of both a controlling and a controlled system. The controlling system interacts with its controlled system using information obtained about the controlled system by various sensors. The state of the controlled system, as perceived by the controlling system, is consistent with the actual state of the controlled system. Hence, periodic monitoring of the controlled system as well as timely processing of the sensed information is necessary. Timing correctness requirements in a real-time system arise from the physical impact of the controlling system's activities upon its controlled system. The most common timing constraints for tasks are periodic, aperiodic and sporadic. A periodic task is one that is activated once every T units. The deadline for each activated instance may be less than, equal to, or greater than the period T. An aperiodic task is activated at unpredictable times. A sporadic task is an aperiodic task with the additional constraint that there is a minimum interval time between task activations. Most of literature categorizes real-time systems into hard real-time systems, software real-time systems, and firm real-time systems [84].

In soft real-time systems, performance is degraded but not destroyed by failure to meet response time constraints. In a hard real-time system, failure to meet response time constraints leads to system failure. In a firm real-time system, some timing correctness requirements are hard, and some are soft [85].

Advances in hardware technology have been able to provide powerful computation throughput. However, the real-time problem is not only fast but is also predictable. On the other hand, the rapid progress in hardware has made distributed computing and multiprocessing a reality, which potentially increases the complexity of real-time software.

2.2.1.1 Complexity Due to Varieties

It is often said that most of the problems associated with developing software are those related to size and complexity. Writing small programs presents no significant problem as they can be designed, coded, maintained and understood by a single person. If that person leaves the company or institution that uses the software, then someone else can learn the program in a relatively short period of time. For these programs, there is an art or craft to their construction and small is beautiful. However, not all software exhibits this most desirable characteristic of smallness. Lehman and Belady [86] reject the simple and perhaps intuitive notion that complexity is simply proportional to the number of instructions, lines of code or modules comprising a program. Instead, they relate complexity to variety and the degree of complexity to the amount of variety. Embedded systems by their nature must respond to real-world events. The variety associated with these events must be catered to; the programs will, therefore, tend to exhibit the undesirable property of complexity. The cost of redesigning or rewriting software to respond to the continuously changing requirements of the real world is prohibitive. Therefore, real-time systems undergo constant maintenance and enhancements during their lifetimes. They must be extensible.

2.2.1.2 Real-Time Facilities

Response time is crucial in any embedded system. Real-time systems are usually constructed using processors with considerable spare capacity [86]. Given adequate processing power, language and run-time support is required to enable the programmer to:

- Specify times at which actions are to be performed,
- Specify times at which actions are to be completed, and
- Respond to situations for which all the timing requirements cannot be met and respond to situations for which the timing requirements are changed dynamically.

When within a system, several tasks are ready to run simultaneously, so scheduling is necessary to make sure that every task can meet its deadline. Most legacy real-time scheduling techniques are based on rate-monotonic analysis (RMA), which originated from the theory of fixed priority scheduling for the problem of scheduling periodic tasks [87], [88]. Later research work has overcome the limitations of the original theory generalized RMA to be practicable for a large range of realistic situations encountered in the design and analysis of real-time systems [89].

2.2.1.3 Manipulation of Real Numbers

In embedded systems, because of the complexity of the plant model and the number of distinct inputs and outputs, most controllers are implemented as computers. The introduction of a digital component into the system changes the nature of the control cycle. The sample and hold operation is carried out by an analog-to-digital converter, in which both converters are under the direct control of the computer.

Within the computer, the differential equations can be solved by numerical techniques, although the algorithms themselves need to be adapted to take into account the fact that the plant outputs are now being sampled. A fundamental requirement for a real-time programming language, therefore, is the ability to express timing constraints and manipulate real, fixed or floating-point numbers with sufficient high resolution for computation.

2.2.1.4 Reliability and Safety

The sheer size and complexity of real-time systems exacerbates the reliability problem; not only must expected difficulties inherent in the application be taken into account, but the same is true for those introduced by faulty software design.

2.2.1.5 Interaction with Hardware Interfaces

The nature of embedded systems requires the computer components to interact with the external world.

2.2.1.6 Efficient Implementation and the Execution Environment

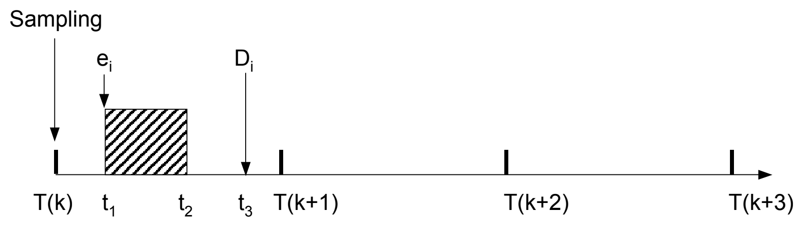
Since real-time systems are time-critical, efficiency of implementation will be more important than in other systems. It is interesting that one of the main benefits of using a high-level language is that it enables the programmer to abstract away from implementation details, and to concentrate on solving the problem at hand [82].

2.2.2 Real-Time Computing Characteristics of Power Conversion Control

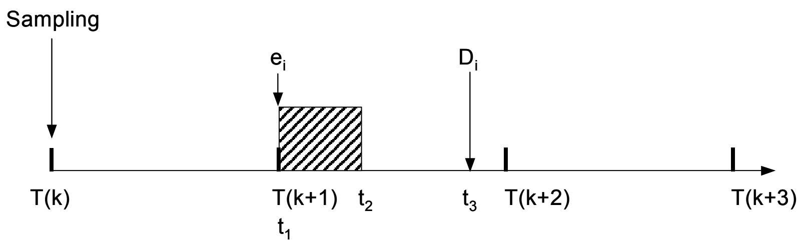
One routine task of the real-time control of a power converter is to generate turn on/off commands to be sent to the switching network. This type of event is periodic and clock-based. During each switching period, stimuli inputs cause the beginning of real-time computing. In the real-time analysis, from the beginning of a real-time computing brought about by a set of stimuli, inputs to the generation of outputs to the same converter with same updating frequency is defined as an event. An event may be further decomposed into multiple actions.

2.2.2.1 Response Time

Figure 2-19 illustrates timing in real-time power converter control. T(n) is the nth switching period. Normally the sampling of feedback signals is synchronized by the switching clock. The kth switching period will be used as an example to show the timing relationships. At T(k), the feedback signals are sampled. After some delay (t_1 – T(k)), the sampled data become the stimuli inputs to event e_i. This delay may be caused by data transferring. Period t_1 to t_2 is the computation time of e_i, and t_3 is the deadline of e_i. Some time period must be saved from t_3 to the beginning of the next switching period in order to transfer the digital control information into real switch turn on/off signals. If the data transfer from the sampling to the digital control is not fast enough, the sampling data at the kth switching period cannot be used for computation during the kth switching period, but can only be used in the next switching period. Figure 2-19 also shows that (a) has a more stringent real-time deadline than (b).



(a)



(b)

Figure 2-19. Real-time models: (a) variable sampling delay and (b) fixed sampling delay.

2.2.2.2 Real-Time Computing

A system based on a switching clock may contain a single control loop or multiple control loops. Within a system, control events for multiple converters may coexist.

A. Single-Rate Event

To control a converter, no matter what control technique is used, the switching network control information must be updated every switching period, while the stimuli inputs may arrive synchronously or asynchronously. As shown in Figure 2-3, the outer control loop might be relatively slow, while the inner current control loop is relatively fast. Each loop is composed of a number of actions. Within a control loop, sequential execution is more commonly seen between actions. Among control loops, parallelism may exist.

For a simple system structure, the sampling data of feedback signals for the slow outer loop and fast current loop may be gathered simultaneously, as synchronized by the switching clock. Thus, the stimuli inputs will be ready at the same time as the computation event. Obviously, in such systems, the outer loop feedback information is over-sampled. An alternative is that the feedback signals can be sampled at a slower rate than the switching frequency, while the feedback of the current loop is synchronized by the switching clock and is ready for computation at every switching period. This means the reference values from the outer loop will not be updated every switching period. This also means the stimuli inputs to an event may arrive at different times. However, no matter what sampling scheme is used, the response to the converter must be generated before the deadline in every switching period.

For the real-time deadline to be met, a control event for a converter should satisfy the following:

$$\sum P_{ij} \leq D_i \, , \tag{2-1}$$

where P_{ij} is the worst-case computation time of action a_j of event e_i, and D_i is the deadline of e_i.

B. Multi-Rate Events

It is very likely that in one power conversion application, the controller needs to control more than one converter, or more generally speaking, handle two or more events at their respective rates. Under this scenario, the multi-rate events model should be used, as described in the following.

- The system is composed of independent events: e_1,.., e_n. Each event corresponds to a control subset with unique external output-updating rates.
- $\forall e_i \in (e_1..e_n)$, T_i, D_i and C_i are predictable.

Based on the above assumptions, any RMA-based scheduling scheme can be applied.

Chapter 3: DISTRIBUTED, MODULAR, OPEN ARCHITECTURE OF POWER CONVERSION CONTROL SYSTEMS

In this chapter, the current practices of control design in power conversion systems are reviewed first. The drawbacks of those methods due to close coupling to hardware will be analyzed. The trend of hardware modularization and standardization in medium- to high- power conversion systems will further increase the complexity of control system design and cost of system maintenance. To handle those problems, a distributed, modular and open control architecture is proposed.

3.1 Current Practices of Control Software Design for Power Converters and the Future Challenges

3.1.1 Current Practices of Control Software Design

In the past, digital control of switching power converters was not widely used. Due to the relatively simple control algorithms and the high switching frequency applied, analog implementation of the control was usually simpler and cheaper. With the development of more complex power converters, such as three-phase converters, PFC circuits, matrix converters, etc., and with the availability of fast and inexpensive digital signal processors (DSPs), A/D and D/A converters, and other digital components, digital control begins to take an important role in power conversion systems, especially medium-to-high-power conversion systems. The way of control software composition has evolved from assembly language based to main-program-and-subroutine, which has been a normal practice academically [14]-[16], [29]-[34]. In recent years, industry began to adopt MatLab ® SimuLink ® and Real-time Workshop ® platform based method, which provides a more systematic way to develop control software [100], [106].

Before the overview of the current practices of control software design in power conversion systems, a simple control application is presented, and this design example will be used through the rest of this chapter for comparison purpose. Figure 3-1 shows the control procedure of a three-phase inverter with current loop closed.

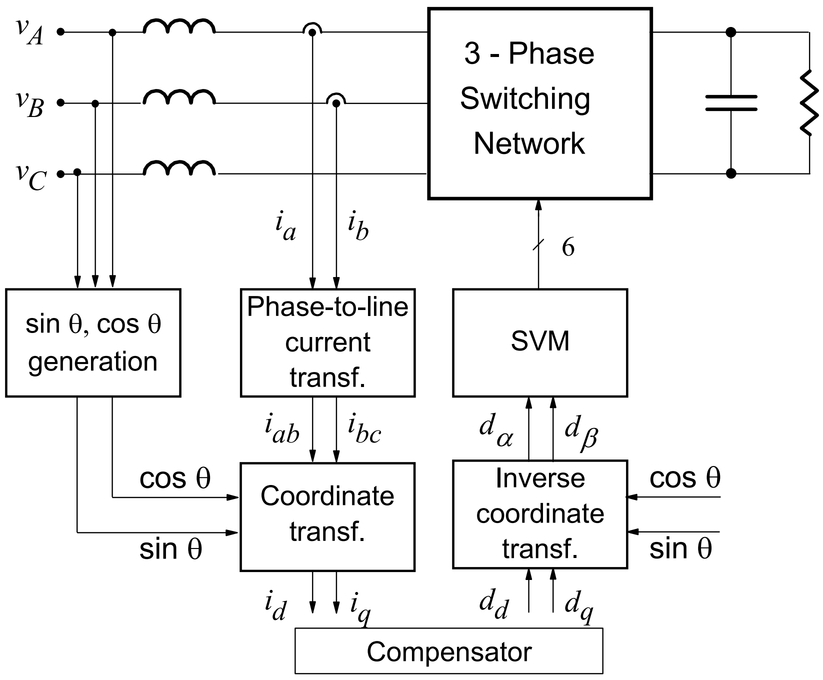


Figure 3-1. Closed-loop control of a 3-phase inverter.

3.1.1.1 Control Software in Assembly

At the beginning of the introduction of digital control into power conversion systems, the emphasis in designing the control software for this application was on computation capacity [29] due to the limitations on the computation bandwidth of digital microprocessors and peripheral components. For that reason, the controller software was written entirely in machine code or assembly code. Table 3-1 shows a sample of assembly code example for a three-phase inverter control. This assembly code is written for Analog Device ADSP 21160.

Table 3-1. Illustration of the assembly control code structure of a 3-phase inverter.

```
….
/*----------------------------- Main Program --------------------------------*/
.SEGMENT/PM pm_code;
Start:
    R0 = EB0WS_2 | EB0WM_01;  { Memory Bank 0 = 2WS, No ack }
    USTAT2 = R0;
    dm(WAIT)=USTAT2;
```

```
            R0 = 0xf000;        { Maximal External Bank Volume }
        dm(SYSCON) = R0;
    ……
    MainLoop:
        Call IRQ1;
        nop;
        Jump MainLoop;          { Waiting for Interrupt }
    /*-------------------------------- IRQ1 Subroutine -------------------------------*/
    IRQ1:
         Call Read_ADC;         { Get the Current Readings from ADC }
         Call get_sincos_theta;
         Call abc_dq;           { Transform Ia, Ib, Ic to dq Reference Frame }
         Call PI_Idq;
         Call dq_to_AlphaBeta;  { dd,dq->dalpha,dbeta }
         Call sector_id;        { Determine the Space Vectors }
         Call Vectors_to_LegDutyCycles;
         Call output_M;         { Output the Space Vectors      }
         rti;
    /*---------------------------- dq to Alpha-Beta Transformation Subroutine ---------------------
*/
    dq_to_AlphaBeta:
         F0 = dm(cos_theta);
         F1 = dm(sin_theta);
         F2 = dm(dd_R);
         F3 = dm(dq_R);
         F5 = F0*F2;    { cos*dd }
         F6 = F1*F3;    { sin*dq }
         F4 = F5 - F6;  { cos*dd - sin*dq }
         dm(dalpha) = F4;
         F5 = F0*F3;    { cos*dq}
         F6 = F1*F2;    { sin*dd}
         F4 = F5 + F6;  { cos*dq + sin*dd }
              dm(dbeta) = F4;
         rts;
```

Though the assembly code is normally optimized to achieve minimal response time, it has several drawbacks:

- Code is barely readable: code written by one designer is hard for others to understand;
- Code is directly coupled to the microprocessor, i.e., it has no transportability; and
- Code is closely coupled to the control host hardware and controlled devices.

Because the assembly code does not provide any level of functionality encapsulation or information hiding, any change in the design decision in the control software will probably necessitate that the entire software be redesigned.

3.1.1.2 Main-Program-and-Subroutine-Styled Control Software in C

As the microprocessor vendors began to provide high-level languages, such as C, and to improve C-complier efficiency, which can compile C code to machine code at an average efficiency of 90%, people began to write digital power conversion control in C in order to achieve better readability and maintainability [92]. Still, because of the concerns related to response time or the real-time performance, the control software is constructed in the simplest main-program-and-subroutine style. Table 3-2 shows the main-program-and-subroutine-styled three-phase inverter control in C language. The software is mainly composed of a main program and a bunch of subroutines. The main program initializes the processor and program after the code is started; this is followed by a dead loop in order to wait for interrupts. For controlling a PWM converter, the interrupt is normally caused by a PWM switching clock. In the interrupt service routine (ISR), the subroutines performing some desired control algorithms are executed one by one. The data transfer between subroutines is through shared memory or global variables.

Table 3-2. A three-phase inverter control in C main-program-and-subroutine.

```
// Constants Definition
…
#include "sin_table.h"
#include "cos_table.h"
…
#define TAB_LEN 672
…
// Global Variables Definitions
….
int index_sin = 0;
int index_cos = 0;
…
float xd, xq;
….
float dd_ref, dq_ref;
….
float d_alpha;
float d_beta;
// Subroutines
…
// Subroutine of Lookups in sin Table
void lookup_sin(void)
{
sin_theta = SIN_TABLE[index_sin];
index_sin ++;
if (index_sin >= TAB_LEN) index_sin = 0;
}
//irq Subroutine
Void Control(void)
{
    read_adc();
    lookup_sin();
```

```
        lookup_cos();
        abc_dq();
        regulator();
        dq_albe();
        modulator();
        output();
    }
    //Main Program
    Main()
    {
      //Microprocessor initialization
      ….
      //Program initialization
      ….
      //Dead Loop
      while (1)
       {}
    }
```

For example, in a three-phase inverter, the switching frequency changes from 10 kHz to 20 kHz; thus a higher resolution lookup sinusoidal table should be used, which also means the table length is changed. The control designer must sweep the previously designed control software to determine the location of the definition of a global constant that is associated with the lookup table length, must understand how this piece of information is interpreted and applied, and then make modifications accordingly.

3.1.1.3 Industrial Platform-Based Software Design

The issue of the close coupling of control software to hardware has gradually gained attention. The optimized process for control development (OPCoDe) project from ABB targets software concepts for hardware independence at certain extensions [100]. OPCoDe uses a hardware abstraction layer to encapsulate hardware dependencies and control board packages to make the control application independent of the control host hardware. In OPCoDe, the code of the control algorithm is generated from MathWorks'® SimuLink® and Real-time Workshop® (RTW) software packages [93], [94], which provide a graphical and integrated model simulation and software development environment. The OPCoDe approach reduces the development period of control software and improves control software code quality. The hardware abstraction layer means that the control software could work with different power hardware products. Because of the features of SimuLink and Real-time Workshop, Function Block Lib is only used during the software development stage, and function blocks are dissolved in the final control code in order to achieve high code efficiency [95]. Table 3-3 shows the structure of the C code generated from RTW Embedded Coder. The generated embedded C code only contains the control algorithm. The interfaces to the specific controlled hardware have to be added manually to the generated C code before it is put into a real hard-oriented system. The lack of support for parallel computation and network communication mechanisms in the original control model in SimuLink also makes this approach difficult for design distributed control applications.

Table 3-3. Pseudo code of embedded coder-generated C program.

```
main()
{
    Initialization (including installation of rt_OneStep as  an interrupt service routine for a
real-time clock)
    Initialize and start timer hardware
    Enable interrupts
    While(not Error)and (time <final time)
      Background task
    EndWhile
    Disable interrupts (Disable rt_OneStep from executing)
    Complete any background tasks
    Shutdown
}

rt_OneStep()
```

```
{
    Check for interrupt overflow or other error
    Enable "rt_OneStep"(timer)interrupt
    ModelStep—Time step combines output, logging, update
}
```

If the control software is not well documented and structured, this work will be time consuming and error prone. Thus, the control software designer might choose to redesign the control software from scratch instead.

DSPACE is another integrated development environment for electric control [96]. The control design is model-based MATLAB® /SimuLink® /Stateflow®, and the production code generator from dSPACE generates the production code straight from MATLAB®/Simulink®/Stateflow®, of which the control software development procedure is very similar to that of OPCoDe. Thus, the generated control code suffers the same problems.

3.2 Power Conversion Systems: Towards Modularization and Standardization

Figure 3-2 shows the hardware structure of traditional, digitally controlled power electronics systems, featuring centralized control and a centralized power stage [26], as shown in Figure 3-2. The design can be customized to meet application requirements. But this type of system lacks standardization and modularization, which leads to a duplicate engineering effort in system development, complex maintenance and high cost.

The PEBB concept was proposed to modularize the power stage [22]. Application manager/hardware manager (AM/HM) architecture was then proposed to divide the control into multiple layers [27]. Figure 3-3 shows the hierarchical structure of PEBB-based power electronics systems. By standardizing the interfaces between each level, standardization and modularization can be achieved. High flexibility in system construction can be achieved under this architecture [28].

In hierarchical power conversion systems, distributed control is very likely to be introduced because of the distributed controlled hardware. The control software will have to not only fulfill the control algorithm, but must also handle communications with distributed hardware and coordination between the distributed software packages. The existing approaches to control software construction have already raised issues in terms of the control design for single, centralized converters. It will be awkward for those approaches to develop control for hierarchical power conversion systems, for which the system structure is becoming more complicated and the control software design and maintenance becomes a more serious concern.

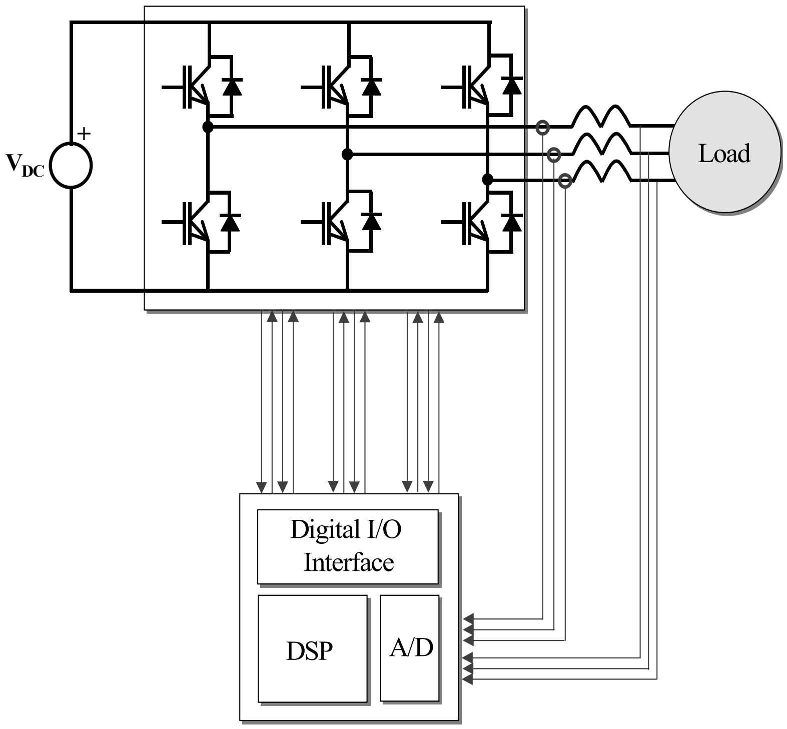


Figure 3-2. Centralized approach to power electronics system hardware design.

3.3 A Real-Time Control System Example: OSACA

Meanwhile, in industrial controls, the same issues appear. In industrial automation, the importance of software is constantly increasing due to the following reasons [97].

1. The production costs for software are smaller, and the software is more flexible as compared to the hardware. In contrast to hardware components, the production costs incurred for software are mainly limited to the development phase.

The increasing percentage of software is due to the development on the hardware sector, as shown in Figure 3-4.

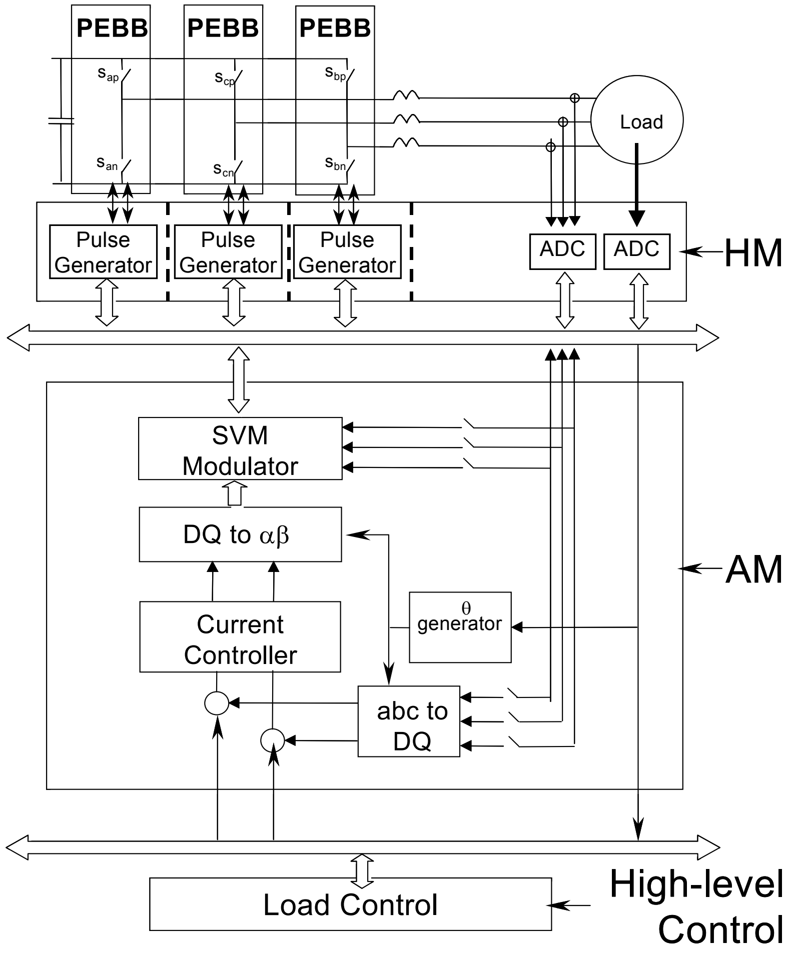


Figure 3-3. Hierarchical power electronics system hardware structure.

This advance and growing importance of software requires more advanced technologies for the software development process than is currently available and applied in industrial controls. Some software systems are still implemented in assembly or C, utilizing only a minor application of software engineering methodologies.

Today's industrial control systems market is dominated by proprietary solutions. The control systems are incompatible, and have only limited communication interfaces. These traditional controllers consist of special hardware and vendor-specific control software (usually offering a more or less standardized application programming interface). The resulting problems are high development costs and long-duration innovation cycles.

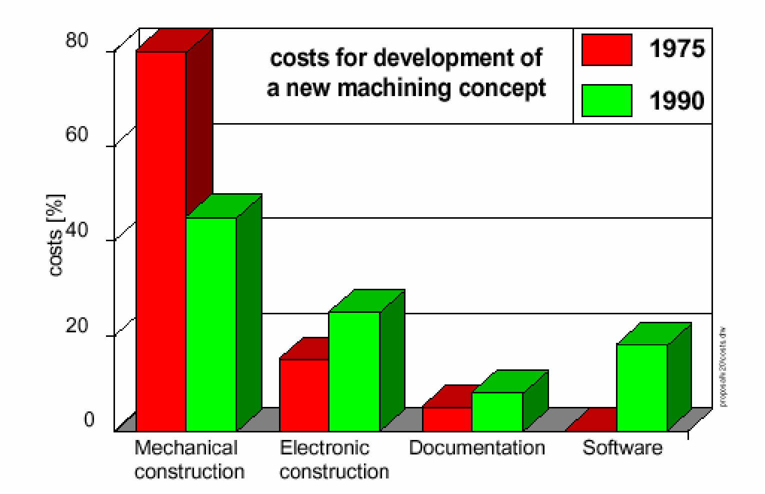


Figure 3-4. Development efforts for machine tool builders[97].

Among various efforts to develop reusable industrial controls, Open System Architecture for Control within Automation System (OSACA) [98] is a notable project, sponsored by several leading companies, such as ATEK, BOSCH, SEIMENS, etc. The objective is the specification of vendor-neutral reference architecture for open control systems, including the functionalities of numerical control for machine tools, robot controls, programmable logic controls, and cell controls. It will form a common base for all kinds of automation systems, and will be open to the integration of new functionalities and for the use of new computing equipment. A reference architecture for application software was specified to ensure the interoperability of application modules, as shown in Figure 3-5.

OSACA has elaborated specifications mainly in the fields of a software reference architecture and a system platform. The reference architecture must enable the interchangeability and extendibility of comparable control-specific application architecture objects (AOs); it defines which architecture objects can be found in a control system, what tasks they perform, and how they interact with each other. To give the vendor of control software the freedom to implement software in his or her own fashion, oriented methods for the definition of externally visible attributes and services are available for each AO. The system platform provides the environment in which the AOs are executed.

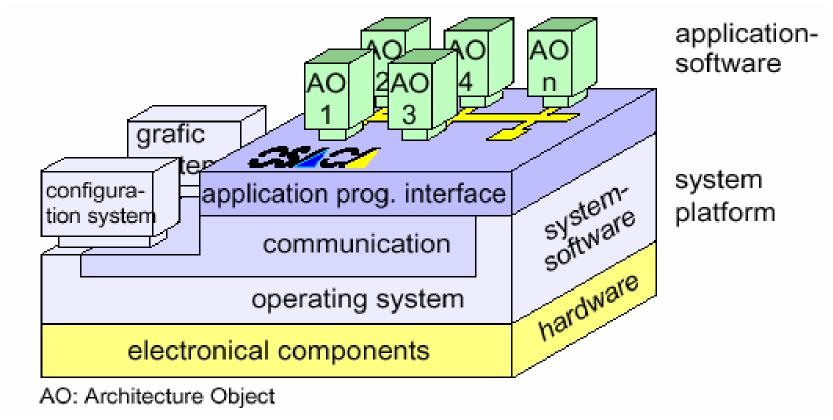


Figure 3-5. OSACA system architecture [98].

From the brief introduction of OSACA reference model, the main design purpose of the model is to separate control from its hardware platform. A question inspired is whether a synergy control reference model is feasible to power conversion systems to handle the software complexity in a systematic way. Furthermore, if such a reference model is viable, how to define and implement those hierarchical levels.

3.4 Interaction between Embedded Control Software and Other Design Areas

From the standpoint of system design, a digitally controlled power conversion system involves several design areas that relate to each other, as shown in Figure 3-6. The application can be seen as a set of requirements that define the behaviors of a specific power conversion system, such as the input/output relationship, the load range and the transient response characteristics. The desired system behaviors are implemented by control and power processing subsystems working interactively. The control subsystem can be seen as control software running on the control host hardware, which is the hardware chip set or digital boards. The power processing subsystem has two levels of interpretations. The power stage topology is the logical circuit description of power processing, and the power stage hardware means all parts of the physical implementation of the circuit, such as switches, passive components, sensing circuits and their interconnections.

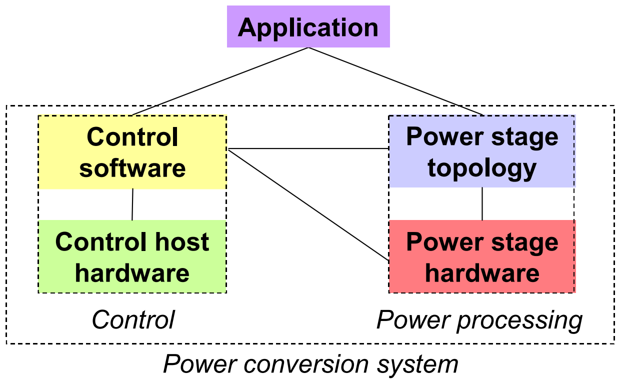


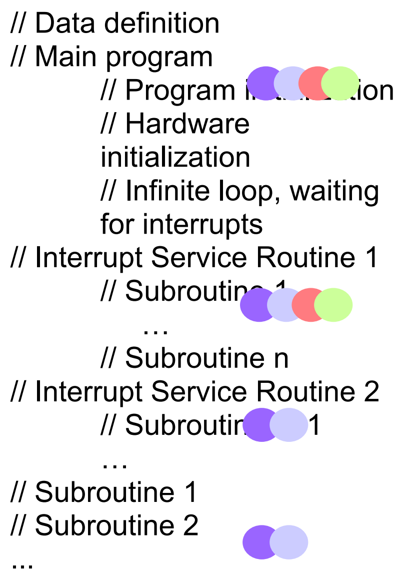
Figure 3-6. Power conversion system structure.

Any change in one or more of the five design areas shown in Figure 3-6 will result in a new system. A three-phase VSI application differs from a three-phase buck rectifier application first in its power stage topology, then in its control algorithm, and finally most likely differs in its power stage hardware. A three-phase VSI with different transient requirements may only affect the selection of the control algorithm or its computation parameters. Even for the same application, using different circuit topologies may require different power stage hardware, and hence different control algorithms and control software. Power stage hardware changes will affect control. For example, using PEBBs requires more peer-to-peer communication and coordination support from the control than using a custom-designed, centralized power stage. Using different control resources also might demand a different programming language and style. Different control algorithms may need different feedback information from the power processing system, hence affecting the physical power stage construction.

These examples of system variations illustrate that the five areas in Figure 3-6 are more or less dependent on each other. Some areas are relatively independent – such as application and control host hardware – while some design areas have strong dependencies on others. Among all five design areas, control software is the most dependent, because, compared to other design areas, control software is the one most flexible to be changed. Thus, because of its ability to adapt to changes in other areas, the selection of control software design is often the last decision to be made within a whole system design procedure.

Normally the selection of control algorithm is dependent on the application, and is decided upon after the power stage topology is chosen. The controlled power stage hardware specifications affect how the control software should communicate with it. The structure of the power stage hardware brings up the requirements for coordination and communication to control the software. The control host hardware sets limitations on control software code size, response speed to external events, and capability of communication with external components.

Since control software has dependencies on application, control host hardware, power stage topology and power stage hardware, if these dependencies are not localized and encapsulated in the control software design, the embedded code will inevitably require significant redesign in order to adapt to even minor changes in a system. The traditional procedural or imperative approach to designing embedded control software results in control software that is hard to maintain and modify, is tightly coupled with other areas of the system, and does not encourage software reusability. Figure 3-7 illustrates in the embedded control software under main-program-and-subroutine style how information dependent on different design areas are interwoven. Changes in one related design area may cause overall redesign of the control software very likely.



```
// Data definition
// Main program
        // Program initialization
        // Hardware
        initialization
        // Infinite loop, waiting
        for interrupts
// Interrupt Service Routine 1
        // Subroutine 1
            …
        // Subroutine n
// Interrupt Service Routine 2
        // Subroutine 1
        …
// Subroutine 1
// Subroutine 2
...
```

Figure 3-7. Information distribution in main-program-and-subroutine style.

Academically and commercially, some research has attempted to tackle some of the dependencies. The OPCoDe [100] project from ABB targets software concepts for control resources and power stage hardware independence. IEC 1131-3 [101] provides a generalized software design model for programming industrial control systems in order to build large systems using equipment from different manufacturers. However, none of these approaches completely achieved localization and encapsulation of every type of dependency.

As mentioned before, both OPCoDe and dSPACE utilize SimuLink/RTW as control design, simulation and generation environment. SimuLink/RTW greatly simplifies the implementation of control algorithm. As shown in Figure 3-8, a control designer can simulate and verify a control algorithm in SimuLink, and then use RTW to generate embedded control code. By selecting an appropriate template, the embedded code can be oriented to different control host hardware.

Figure 3-9 shows an example of SimuLink model of closed-loop control for a three-phase inverter. This model embraces modularity design. Any function block can be added into the system by drag-and-place. However, this model only describes the control algorithm. The automatically generated embedded C code from this model has to be tailored manually for specific hardware. It is also hard to describe distributed control in this model, and how to handle communication over networks is also not provided. To some extent, most of the control composition relies in handling the interfaces to various types of hardware.

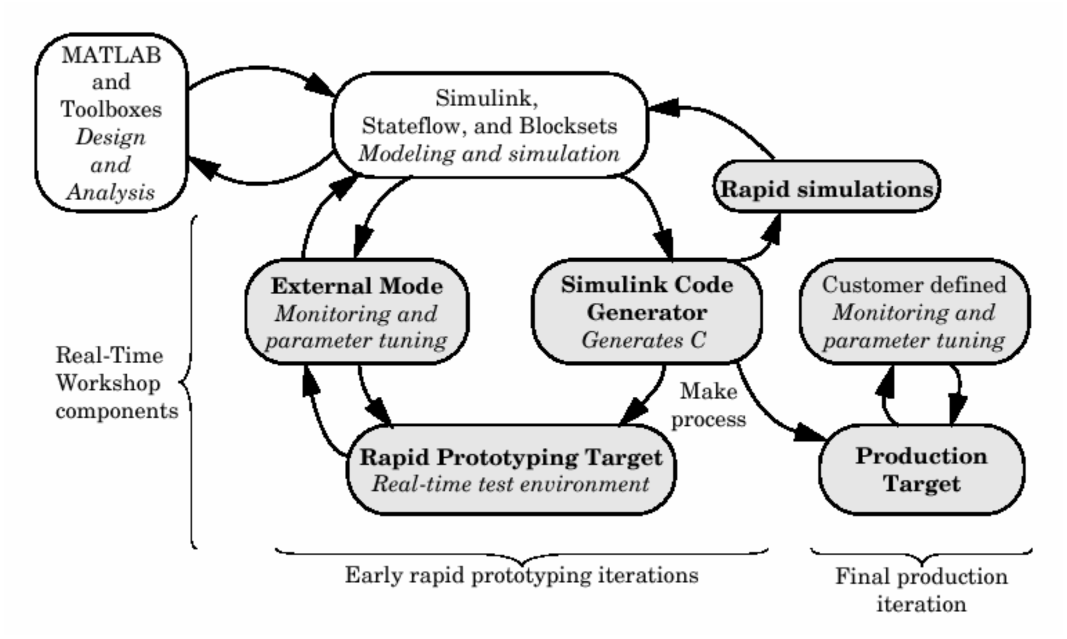

MATLAB and Toolboxes *Design and Analysis*
Simulink, Stateflow, and Blocksets *Modeling and simulation*
Rapid simulations
External Mode *Monitoring and parameter tuning*
Simulink Code Generator *Generates C*
Customer defined *Monitoring and parameter tuning*
Real-Time Workshop components
Make process
Rapid Prototyping Target *Real-time test environment*
Production Target
Early rapid prototyping iterations
Final production iteration

Figure 3-8. Simulation and automatic code generation in MatLab SimuLink/RTW [106].

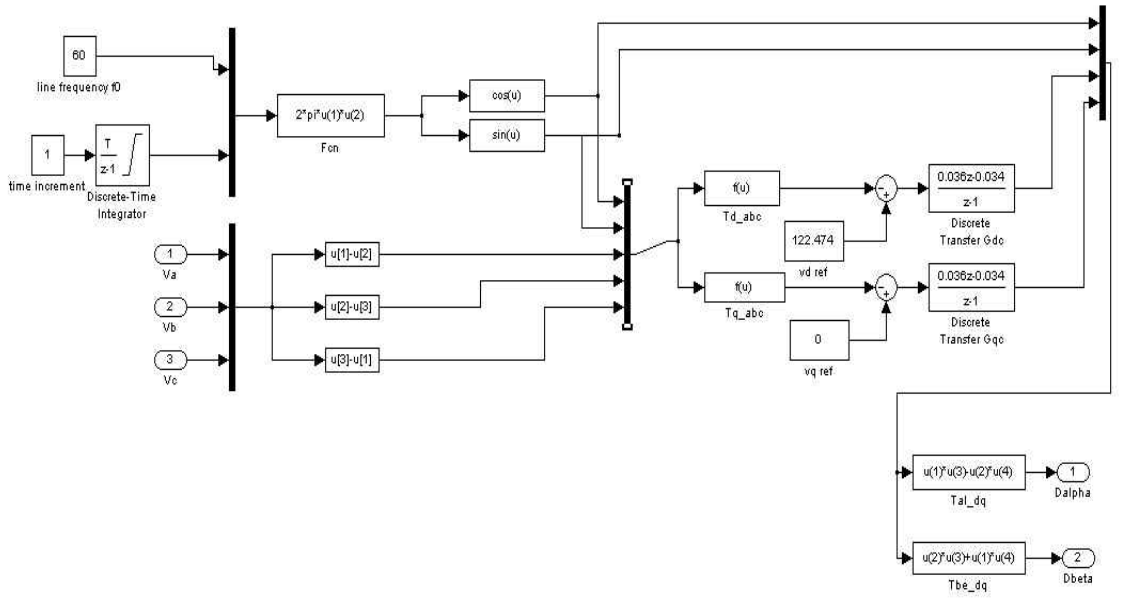


Figure 3-9. SimuLink model of a closed-loop 3-phase inverter control.

Figure 3-10 shows an example of localization and encapsulation of controller device dependencies. Modulator is normally the last stage of a PWM control algorithm, and is the part most likely to be related to power stage hardware specifications. If the modulator talks to the power stage directly, the modulator will become dependent of power stage hardware, such as timing requirements from gate drivers and turn on/off characteristics. Changes in power hardware are very likely to cause the whole modulator to be redesigned. However, if there is another layer added between the modulator and power stage, which serves as device driver. At the interface to modulator, the device drive takes the duty cycles as inputs. According to the specifications of power hardware in the system, the device driver is responsible to translate the duty cycle commands into hardware specific switching pulses. By this approach, the functionality of modulator can be concentrated on modulation scheme and the outputs can be standardized to duty cycles. On one hand, new modulation is easy to plug into the control software, as shown in Figure 3-10 (b) and (c). On the other hand, changes in the power hardware will not cause significant redesign in the modulator, instead different device drivers will be used to incorporate those changes, as shown in Figure 3-10 (d).

3.5 Distributed, Modular, Open Control Architecture of Power Converters

Based on the analysis of how control software relates to other areas in power conversion system design, Figure 3-11 shows a general control software model, which is capable of mapping and localizing different types of dependencies. In this general model, the control software is constructed as two main layers: control application and runtime environment support. The control application encapsulates all the dependencies related to the system application. The runtime environment support makes all the non-application-related dependencies transparent to the control application.

It should be noticed that although the choice of control algorithm is dependent on the application or power stage topology, it could be implemented from a set of commonly used functionalities, which can be repeated in different applications. Function Block Lib is a library of such independent function modules, which makes it possible to construct a control algorithm from a fairly large piece of reusable code instead of instruction lines. The control algorithm abstraction layer captures all dependencies used to form a control algorithm, such as the collection of function blocks that will be used, the pattern of those function blocks being connected, and computation parameters, etc. The power stage hardware abstraction layer encapsulates power stage hardware specifications, such as interpretation of digital control information to analog control signals and analog feedback information to digital numbers.

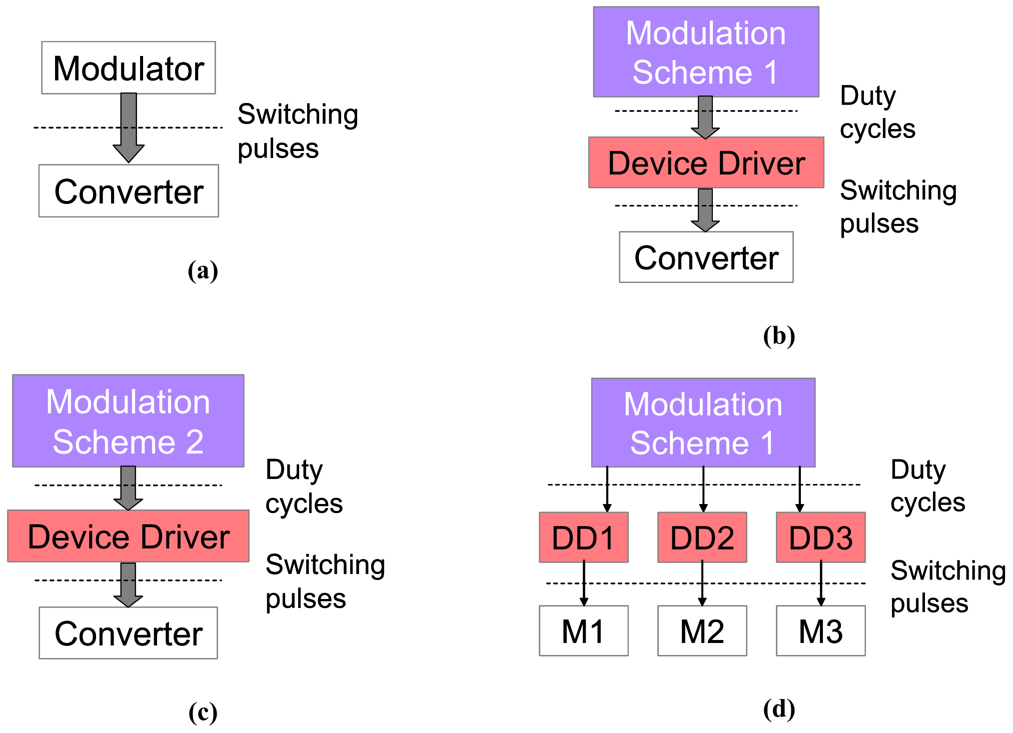


Figure 3-10. Localization and encapsulation of dependencies of controlled devices.

Control applications (CAs) will work smoothly on the host hardware with the runtime environment support layer. The CA run-time support provides API to other layers and necessary real-time support for internal interactions within the control application. Using the network interface, the control software can communicate with other control intelligence in a distributed control environment. Through the host interface, control software can be easily migrated to different host hardware.

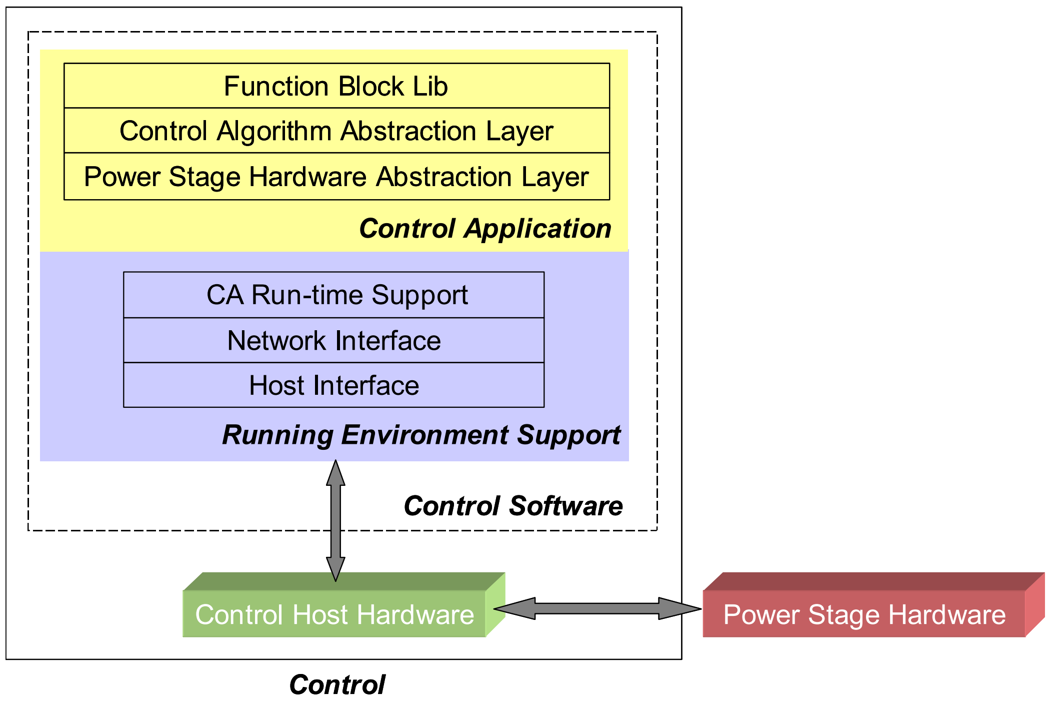


Figure 3-11. General control software model.

If it is constructed under the general model shown in Figure 3-11, the control software should be reusable, reconfigurable, and easy to maintain. Commercially and academically, some research efforts have applied software engineering techniques in industrial control software design, aiming for fast software development or reusable software design. However, most of them only implement part of the general software model described in Figure 3-11. Next, the mainstream research and practice on software development in power conversion systems will be mapped into the general model, and a full implementation of the general model will be presented.

Chapter 4: IMPLEMENTING CONVERTER CONTROL BY APPLYING DATAFLOW STYLE

In this chapter, an implementation applies a dataflow style to the open control architecture proposed in the previous chapter. Several software styles will be compared to show the reason that dataflow is suitable for power conversion control.

4.1 Software Architecture

Software architecture is emerging as an important discipline for engineers of software. The issues of the design and specifications of the overall system structure become more significant than the choice of algorithms and data structures used in the computation.

4.1.1 Basic Concepts of Software Architecture

Structural software issues include the organization of a system as a composition of components; global control structures; protocols for communication, synchronization, and data access; allocation of functionality to design elements; composition of design elements; physical distribution; scaling and performance; and selection among design alternatives. This is the architecture level of software design. Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components [90].

4.1.2 Software Styles

One of the hallmarks of architecture design is the use of idiomatic patterns of system organization. The use of patterns and styles of design is pervasive in many engineering disciplines. Software has organizational styles, too. At the architectural level, these are often associated with phrases such as a client-server system, a pipe-filter system, or a layered architecture. Some of these are commonly associated with specific design methods and notations, such as object-oriented (OO) and dataflow organizations. Other architectural patterns are associated with specific classes of systems, such as the traditional organization of a compiler, the International Standard Organization's Open

System Interconnection Reference Model, and common patterns for OO design. Figure 4-1 shows a partial list of some of the principle categories.

Dataflow Systems
- Batch Sequential
- Pipes and Filters

Call-and-Return Systems
- Main Program and Subroutine
- OO Systems
- Hierarchical Layers

Independent Components
- Communicating Process
- Event Systems

Virtual Machines
- Interpreters
- Rule-Based Systems

Data-Centered Systems (Repositories)
- Databases
- Hypertext Systems
- Blackboards

Figure 4-1. A partial list of common architecture styles.

A software style treats a specific system as a collection of computational components together with a description of the interactions among them – they are connectors. Examples of components are clients, servers, filters, layers and databases. Examples of connectors are procedure call, event broadcast, database protocols and pipes. Next, we will give several examples of software styles, such as pipe-and-filter, OO, implicit invocation and layered system.

In a pipe-and-filter style, as shown in Figure 4-2, each component has a set of inputs and a set of outputs; it reads streams of data on its inputs, and produces streams of data on its outputs. The components, termed as filters, apply a local transformation to the input streams and compute incrementally. The connectors, termed as pipes, serve as conduits for the streams.

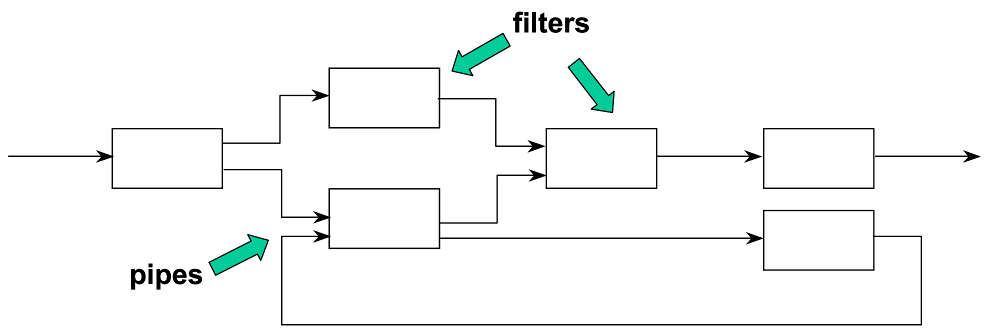


Figure 4-2. Filters and pipes.

The pipe-and-filter style has the following merits:

- It is easy for designers to understand the overall input/output behavior of a system as a combination of behaviors of individual filters;
- It supports reuse;
- The systems are easy to maintain and enhance: New filters can be added to existing systems and old filters can be replaced; and
- It naturally supports concurrent execution.

The major problem of pipe-and-filter systems is their limited ability to handle interactive applications.

Main-program-and-subroutine style falls into call-and-return systems category. Figure 4-3 shows a generic structure of this type of software system. OO is another well known software style in this category. In the style based on data abstraction and OO organization, data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are objects responsible for preserving the integrity of a resource. Objects interact through function and procedure invocations. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents. The most significant disadvantage of OO systems is that in order for one object to interact with another, it must know the identity of that object.

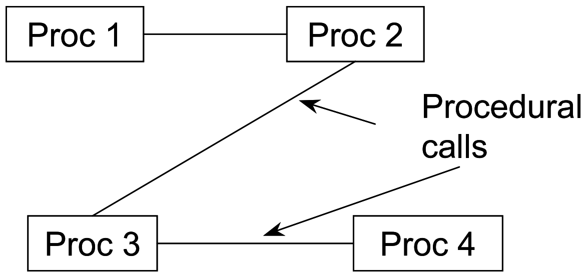


Figure 4-3. Call-and-return system.

An alternative to explicit invoking among components, as exists in OO systems, is implicit invocation, as shown in Figure 4-4. This style has historical roots in systems based on actors, constraint satisfaction, daemons and packet-switched networks. In an implicit invocation system, a component can announce or broadcast one or more events. Other components can register an interest in an event by associating a procedure with it. One important benefit of implicit invocation is that it provides strong support for reuse. The primary disadvantage is that components relinquish control over the computations, since they are performed by the system.

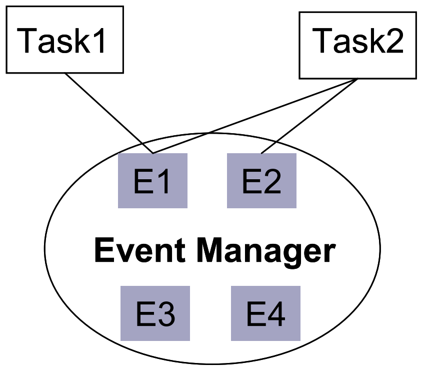


Figure 4-4. Implicit invocation.

A layered system, as shown in Figure 4-5, is organized hierarchically, with each layer providing service to the layer above it and serving as a client to the layer below. The connectors are defined by the protocols, which determine how the layers will interact. The best-known example of this style is the ISO-OSI reference model. Layered systems have some desirable properties, such as offering support for increasing levels of abstraction, support for enhancement, and the ability to be reused. The main disadvantages of the layered system are the difficulties related to both structuring a system into a layered fashion and to finding the proper levels of abstraction.

Different software styles will provide different benefits. A choice of component and interaction patterns and styles reflects a decomposition of a problem domain. Thus, a proper choice of software style is important to long-term success for a software system.

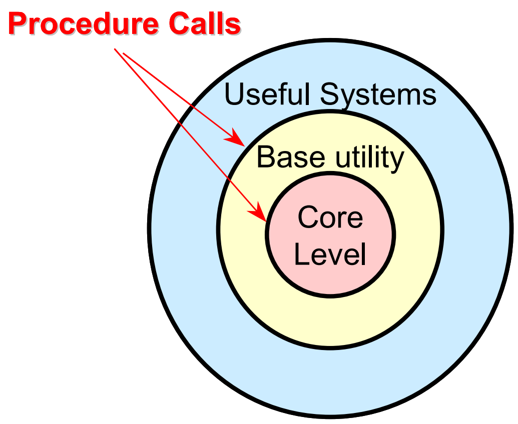


Figure 4-5. Layered system.

4.2 Requirements for Software Style

The basic concepts of software architecture show that any software style has its fundamental patterns of components and interactions. Different software styles could offer different benefits. As revealed by the analysis of real-time embedded control for medium- to high-power conversion systems in Chapter 2, several considerations are important in the selection of a suitable software style to implement the proposed distributed, modular, open control architecture.

Firstly, for control software design, it is desirable that under the applied software style, the expression and composition of control software can naturally fit the understanding of the problem to be solved. This will save the software designer tremendous time in mapping the problem domain into the software paradigm. The objective of digital control of power conversion systems is to control the power converter behaviors as designed by manipulating the input data through certain math functions or logic functions in order to generate control signals. The chosen software style should be able to naturally map control schemes used in the power conversion application, as describe before.

Secondly, because modularity is a main means to reduce the system construction effort in the proposed open control architecture, the applied software style should be able to support the implementation of certain functionality at module level. Besides functionality encapsulation, the component style should also be able to separate functionality from its implementation, such that the pure internal modification of one module will not affect other modules in the system. This requires that the component style have the capability of good data abstraction and information hiding.

Thirdly, support for distributed computation is also an important merit, since modular and distributed power processing hardware is the trend. This requires that the control software be easy to decompose and allocate into a distributed computing environment. This also means that the interaction pattern should render components independent of each other so that reallocation of one component will not cause an overall redesign. Since parallel processing will improve the performance of control in a distributed computing environment, it is beneficial if the applied software style support parallel execution between components.

Lastly, because of the real-time nature of power conversion control, facilities must be provided for real-time scheduling under the applied software style, and the resulting control software must maintain high execution efficiency.

4.3 Comparison of Software Styles

A power electronics control system is a real-time system, which has restricted resources and a set of deadlines to meet. As distributed control architectures become commonplace for power electronics systems, the size and complexity of the corresponding control software will increase. Which software style will be best suited to implement the open control architecture presented in Chapter 3. Table 4-1 gives comparisons of several candidate software styles, using the main considerations discussed in Section 4.2. Table 4-1 shows that the dataflow style might be a good candidate.

Table 4-1. Comparison of software styles.

	Accurate and Easy Map to Control Algorithm	Components of Well-Defined Interface	Easy to Reallocate	Efficient Execution
Dataflow	+	+	+	+-
Main-Program-and-Subroutine	+	-	-	-
Implicit Invocation	-	+	+	+-
Layered System	-	+	-	-

A dataflow approach to software construction provides five distinct benefits. First and foremost, the focus on constructing applications from highly independent computational units that know nothing of their "neighbors" in an application enables software reuse. Reuse requires one to develop software parts that can be composed in different ways so that they are useful in a large number of applications. This is only practical when such parts are loosely coupled—they depend as little as possible on the other parts with which they interoperate. Otherwise, one must reuse an entire assembly as-is, or not at all.

Second, dataflow enables a new approach to application reconfigurability. While many software programs can be thought of as a graph, dataflow applications actually are graphs. When viewing a program this way, one simple view of "reconfiguration" is the addition or removal of nodes or edges in the graph. These operations are typically easy to draw, but very difficult to implement for main-program-and-subroutine applications. For dataflow applications, additions and removals are exceedingly simple. This is primarily the result of the requirement that processes be completely independent.

Third, a dataflow approach provides a natural mapping onto distributed software execution. Because processes only communicate by sending messages over data channels, it makes little difference whether two processes are executing on the same processor or on different processors. Data channels can be implemented simply as shared variables for processes that are allocated to the same processor or through some form of networked message transmission for processes allocated to different processors. Further, with effective processor assignment algorithms, one application can be easily mapped onto a number of distributed topologies without ever modifying the dataflow graph or the implementation of any dataflow processes.

Fourth, by encouraging reuse and focusing the application designer's attention on reusability, a dataflow approach fosters the construction and population of a library of reusable components. A domain analysis of power electronics control problems will be instrumental in identifying the common, recurring processing tasks that are most suited for such a library.

Finally, such a library provides ideal support for rapid application development. A well-populated library will allow the routine aspects of a control algorithm to be pieced together easily from pre-written and pre-tested parts. Instead, the application designer can focus her attention on the unique and most difficult aspects of the system at hand.

Since the dataflow approach presents balance solution among the requirements towards implementation of the proposed open control architecture, it will be chosen as software style to map the distributed, modular and open control architecture for power conversion systems.

Chapter 5: DATAFLOW INFRASTRUCTURE

This dissertation proposes dataflow architecture as a complete implementation of the open control architecture shown in Figure 3-11, in order to design reusable and reconfigurable control software for power conversion systems. Dataflow is a software design technique with a long history in computer science [102]-[104]. In the dataflow style, a control application is implemented as a set of concurrently executing processes. Dataflow processes communicate by sending messages through one-way message queues called data flows or channels. Each dataflow process is independent, and knows nothing about the other processes in the application—it merely consumes data from some channels and produces results on other channels.

Figure 5-1 shows the dataflow architecture. In the dataflow implementation of embedded control for power conversion systems, a control algorithm is implemented as a set of concurrently executing processes, which we call elementary control objects (ECOs) in order to reflect their role as the building blocks of larger control applications. ECOs communicate through one-way message queues called data channels. Each ECO is independent, and knows nothing about the other ECOs in the application—it merely consumes data from some channels and produces results on other channels. Dataflow computing is reminiscent of signal filtering and processing, and leads one to design ECOs that are modular and reusable. Constructing control applications then becomes the process of picking ECOs from a library and "plugging them together" into the desired pattern. ECOs protect independent functionalities that can be reused in different control applications. The dependency of the control algorithm is encapsulated in the dataflow graph, and can only affect that dataflow graph. In other words, the changes in the control algorithm, caused by either application requirements or the power stage topology, will be adapted in the dataflow graph. A driver ECO is especially written for the control software to communicate with a specific piece of controlled hardware. All the hardware-related information is hidden inside the driver ECO. Using a new piece of hardware will only cause the corresponding ECO driver to be substituted. The dataflow architecture real-time kernel (DARK) supports lightweight process management, data channel management, system resource allocation, device driver support, and interrupt handling. DARK also provides a unified interface to control host hardware and a communication network to other control software.

Using the dataflow approach allows reusable and reconfigurable designs for control software that could “plug-and-play” in different power conversion applications and systems.

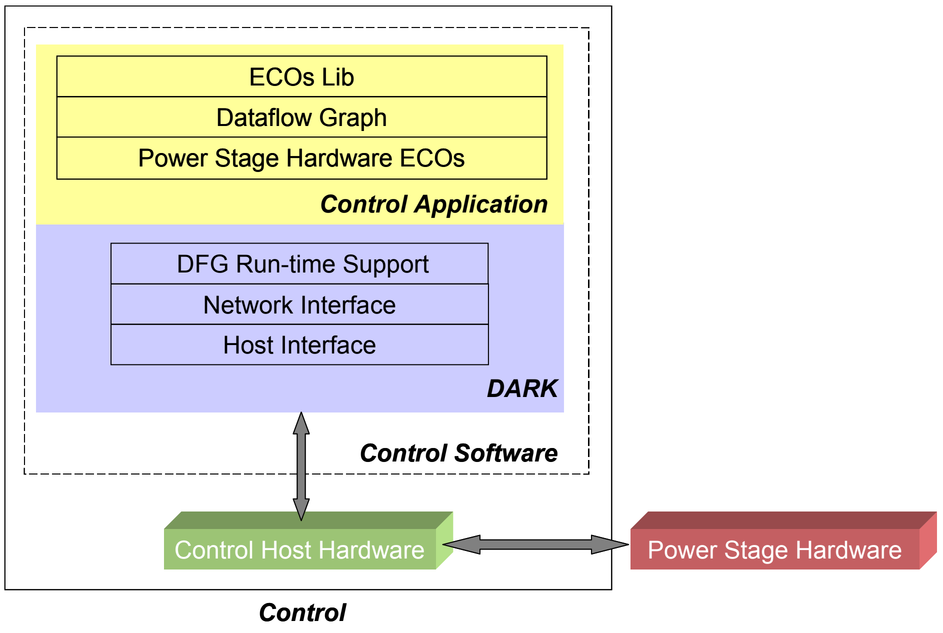


Figure 5-1. Dataflow software architecture.

Software written using dataflow architecture consists of a collection of independent components running in parallel that communicate via data channels; such a design can be succinctly depicted graphically, as shown in Figure 5-2, in which each node is a computational component, and each arrow is a buffered data channel. Each concurrently executing node is a self-contained software part with well-defined behavior. Data channels provide the sole mechanism by which nodes can interact and communicate with each other, ensuring minimal coupling and greater reusability. Data channels can also be implemented transparently between processors to carry messages between components that are physically distributed. Choosing this component model for embedded control software alleviates many of the negative aspects of the more traditional main-program-and-subroutine organization. More importantly, however, it also opens up the possibility of developing a library of commonly recurring, standardized control software functions encapsulated in reusable dataflow components. Such a design library would allow a new control application to be configured rapidly from an existing collection of components.

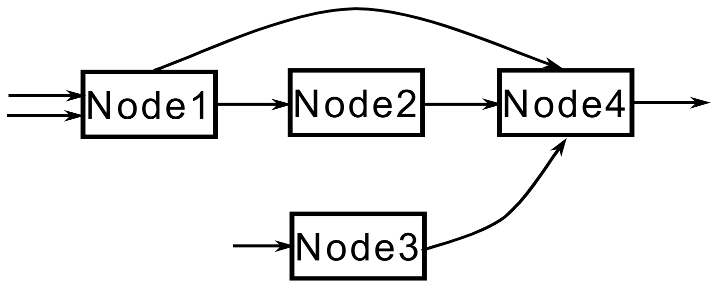


Figure 5-2. Dataflow software.

5.1 Design Concerns Regarding Dataflow Approach

Despite all of these benefits, dataflow techniques have seen little use in embedded software domains. This is because of 3 significant concerns about dataflow applications.

First, the costs of managing concurrent execution of dataflow processes might be too prohibitive. If dataflow processes are implemented as concurrently executing threads, there will be significant additional overhead associated with managing these threads, scheduling them, and switching between them.

Second, the costs of managing data channels might be too prohibitive. All interaction between processes involves sending messages on data channels. No other communication is permitted. This communication involves one process writing data, and another reading the data, plus the necessary inter-process synchronization. The total cost for managing such an interaction is likely to be more expensive than simply transferring parameters to a called subroutine.

Third, the use of a dataflow style raises questions about the proper "granularity" or size of the computational processes—how "big" each process is, in terms of the amount of work it does. A dataflow approach may lead one towards a design where a larger number of simpler, more reusable processes are used. This may lead to an unacceptable cost in terms of the extra overhead needed to manage a larger number of processes. Instead, one would prefer to have fewer processes where possible.

Carrying these concerns, the following sections will present detailed design of the dataflow architecture.

5.2 Elementary Control Objects

Each ECO manipulates the input data that it receives according to its behavior, generating output that can be connected to other ECOs. There are no explicit calls between ECOs—in fact, an ECO has no knowledge of the other nodes the system contains, or of the identities of the other nodes with which it communicates. This inherent independence allows ECOs to be treated as concurrently executing, active objects. This natural concurrency, together with the ability to transparently map data channels across physical component boundaries, provides a direct and simple mechanism for supporting the distributed control of power electronics systems.

An ECO contains input and output data channel descriptions, a startup parameter description, and an implementation, as illustrated in Figure 5-3. Three distinct types of ECOs exist within the embedded control domain—computational ECOs, coordination ECOs and driver ECOs. A computational ECO embodies some specific computational behavior needed for an application. Figure 5-4 shows several examples of computational ECOs. A coordination ECO, on the other hand, is designed to support the transparent management and control of distributed system hardware assets, as illustrated in Figure 5-5. Driver ECOs encapsulate hardware dependencies, and provide a standard program interface to control hardware. Figure 5-6 shows an A/D driver ECO.

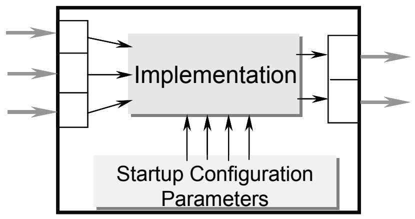


Figure 5-3. ECO structure.

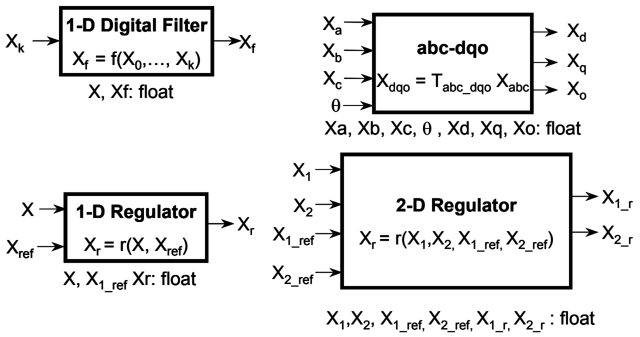

1-D Digital Filter
X_k
$X_f = f(X_0,\ldots, X_k)$
X_f
X, Xf: float
abc-dqo
X_a
X_b
X_c
θ
$X_{dqo} = T_{abc\_dqo} X_{abc}$
X_d
X_q
X_o
Xa, Xb, Xc, θ , Xd, Xq, Xo: float
1-D Regulator
X
X_{ref}
$X_r = r(X, X_{ref})$
X_r
X, X_{1_ref} Xr: float
2-D Regulator
X_1
X_2
X_{1_ref}
X_{2_ref}
$X_r = r(X_1,X_2, X_{1\_ref}, X_{2\_ref})$
X_{1_r}
X_{2_r}
$X_1,X_2, X_{1\_ref}, X_{2\_ref}, X_{1\_r}, X_{2\_r}$: float

Figure 5-4. Examples of computational ECO.

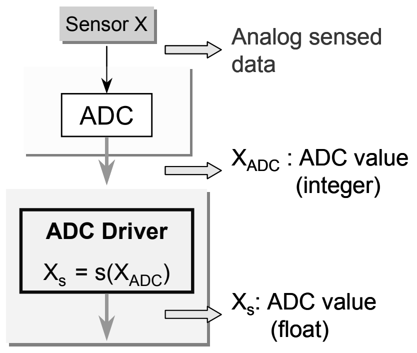


Figure 5-5. Example of a coordination ECO.

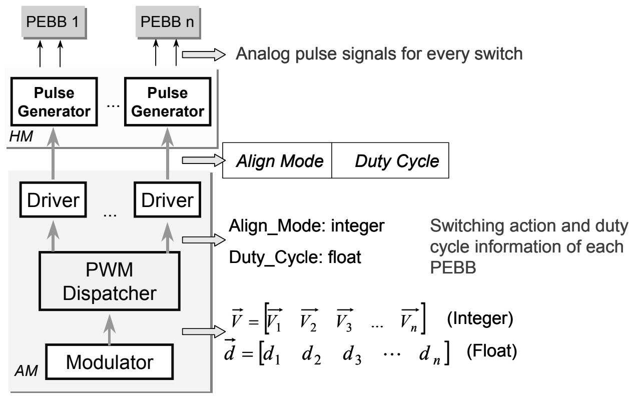

PEBB 1
PEBB n
Analog pulse signals for every switch
Pulse Generator
Pulse Generator
HM
Align Mode
Duty Cycle
Driver
Driver
Align_Mode: integer
Duty_Cycle: float
Switching action and duty cycle information of each PEBB
PWM Dispatcher
$\vec{V} = [\vec{V_1}\ \vec{V_2}\ \vec{V_3}\ \dots\ \vec{V_n}]$ (Integer)
$\vec{d} = [d_1\ d_2\ d_3\ \cdots\ d_n]$ (Float)
AM
Modulator

Figure 5-6. A/D driver ECO.

ECOs are data-driven entities. They operate on data arriving on incoming data channels whenever possible, producing their results on outgoing data channels. As a result, most ECOs are implemented as loops that wait for incoming data and then process that data when it arrives. Table 5-1 shows a typical pseudo-code skeleton for an ECO implementation. At the beginning of the main loop making up the ECO's body, the wait_to_fire() API function is called. This function checks whether the data required by the ECO is available, and if not, suspends the ECO until the necessary data arrives. Incoming data expectations are expressed through firing rules that tell the underlying RTOS which data channels must contain data in order for this ECO to be ready to perform some processing. If any firing condition is satisfied, an action will be taken according to the firing condition.

An ECO can have multiple actions corresponding to multiple input firing conditions. Within each action, however, execution procedures are quite similar—reading inputs from input data channels, performing computation, and writing outputs to output data channels. The most commonly used API functions within an ECO are wait_to_fire(), read_DC() and write_DC().

Information about each ECO process is maintained by the RTOS in a process data block, illustrated in Table 5-2. This block of information allows an ECO to easily access its input and output data channels and provides other information related to RTOS system calls.

Table 5-1. Pseudo-code for a typical ECO implementation.

```
void Sample_ECO_Body ( Process_Data* p ) {
  while ( wait_to_fire( p ) ) {
    switch ( p->wakeup_call ) {
      case SAMPLE_ECO_FIRING_MASK_1:
        read_type_DC( p->inport[X1],  &x1 );
        …
        /* -- action 1 --*/
        …
        write_type_DC( p->output[Y1],  y1 );
      break ;

      case SAMPLE_ECO_FIRING_MASK_2:
        read_type_DC( p->inport[X2],  &x2 );
        …
        /* -- action 2 --*/
        …
        write_type_DC( p->output[Y2],  y2 );
      break ;
          default: break;
    }
  }
}
```

Table 5-2. Process data structure.

```
typedef struct
{
   Data_Channels        in_port;          /* Input port handles.           */
   Data_Channels        out_port;         /* Output port handles.          */
   ECO_Configuration config               /* ECO-defined block of          */
                                          /* configuratation parameters.   */
   Firing_Mask          wakeup_call;      /* The last firing mask used to  */
                                          /* wake up a sleeping ECO.       */
   Priority             current_priority; /* The ECO's current priority;   */
                                          /* it should only be changed     */
                                          /* using set_priority().         */
} Process_Data;
```

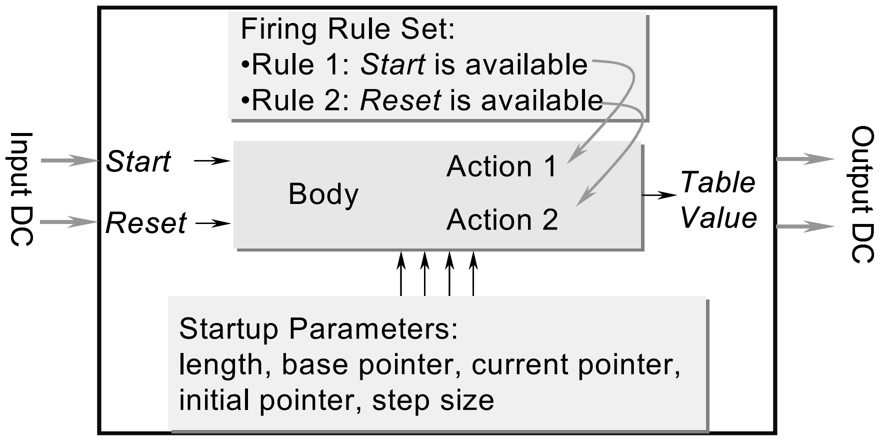

Firing Rule Set:
•Rule 1: *Start* is available
•Rule 2: *Reset* is available
Input DC
Start
Reset
Body
Action 1
Action 2
Table Value
Output DC
Startup Parameters:
length, base pointer, current pointer, initial pointer, step size

Figure 5-7. Lookup_table ECO.

As an example, consider the Lookup_table ECO from Figure 2. This ECO provides a simple behavior that is easy to understand, but at the same time shows the typical internal structure of a dataflow software component written in this style. Figure 5-7 shows the internal view of the Lookup_table ECO. This ECO maintains a circular table. Every time it gets triggered, the Lookup_table ECO outputs the current table value, and adjusts its table pointer to the next table position. **Error! Reference source not found.** shows the configuration block of the Lookup_table ECO. Besides incoming data, an ECO may depend on some other parameters to implement its functionality. These kinds of data are stored as parameters for each ECO, and kept access pointer in the process data block of each ECO. For example, for the Lookup_table ECO, the necessary computation parameters are: table length, table base pointer, initial table pointer, current table pointer, modification step size, etc., as shown in Table 5-3.

Table 5-3. Configuration block of Lookup_table.

```
typedef struct
{
        int Table_Length;
        int Step;
        float *Table;
        int Index;
        int Index_init;
} Lookup_table_Configuration;
```

The ECO's implementation, also called its body, is the program code that describes its behavior. Table 5-4 shows the implementation for the Lookup_table ECO.

An ECO may have different kinds of responses to different input patterns. A firing mask defines an input pattern that an ECO may respond to—a simple bit vector that indicates which incoming data channels must hold data to trigger this response. A priority firing mask is a mask that is paired with a priority value. If the associated mask is triggered, the associated priority will used to reset the ECO's process priority as it responds. This allows ECOs to run at one priority for "normal" events but automatically increase their priority when a critical event requires a response. Whether one or many firing masks are used, as well as the priorities assigned to each, can be determined by the application designer.

Table 5-4. ECO body of Lookup_table.

```
void Lookup_table_Body( Process_Data* p ) {
  float data;
  bool cmd;
  Lookup_table_Configuration* config =
    (Lookup_table_Configuration*) ( p->config );

  while ( wait_to_fire( p ) ) {
    switch ( p->wakeup_call ) {
      case LOOKUP_TABLE_FIRING_MASK_RESET:
        read_bool_DC( p->in_port[LOOKUP_TABLE_RESET], &cmd );
        config->Index = config->Index_init;
        data = config->Table[config->Index];
        config->Index += config->Step;
        if ( config->Index >= config->Table_Length )
          config->Index = 0;
        write_float_DC( p->out_port[LOOKUP_TABLE_DATA], data );
        break;

      case LOOKUP_TABLE_FIRING_MASK_START:
      default:
        read_bool_DC( p->in_port[LOOKUP_TABLE_START], &cmd );
        data = config->Table[config->Index];
        config->Index += config->Step;
        if ( config->Index >= config->Table_Length )
          config->Index = 0;
        write_float_DC( p->out_port[LOOKUP_TABLE_DATA], data );
        break;
    }
  }
```

A complete firing rule for an ECO is an array of priority firing masks defining all the combinations of input patterns that an ECO might respond to in an application. For example, the Lookup_table ECO will respond with one action if “Start” data is present, and respond with an alternate action if *Reset* data is present. When the firing rule is indicates the availability of input on the “Start” data channel, the Lookup_table ECO outputs the value in the table indicated by the table’s current pointer calculated from parameters in the Lookup_table configuration. Then the current pointer is adjusted according to the step size in the configuration. This entire behavior is encapsulated in a code segment within the Lookup_table ECO body, as shown in Table 5-4. If *Reset* data is available, it will cause another action to be executed. In this action, shown in Table 5-4, the table current pointer will be reset to an initial value and the initial table value will be output. The ordering of masks within the firing rule determines which takes precedence. Table 5-4 shows the body of Lookup_table ECO. If input “Start” is available, action reset gets executed; if input “Reset” is available, action increment gets executed.

In our design, an ECO can only interact with outside through the ECO API. As long as the API kept intact, the change in the ECO body will not affect other ECOs. Thus, ECOs are independent from each other. An application can use an ECO flexibly through firing rules.

5.3 Data Channels

Data channels serve as the sole communication paths connecting ECOs into a cohesive control algorithm. Each data channel connects a pair of ECOs: The source ECO generates data, and the sink ECO consumes data. Note that data channels are unidirectional—data can only flow from one source ECO to one sink ECO. Data channels carry typed data based on the application requirements; strong typing helps certain kinds of interconnection errors to be discovered during development, rather than later during operational testing. Each data channel has a data queue for buffering the data between ECOs that operate at different speeds. When developing the overall software structure, the application designer configures each data channel’s data type, buffer size, source connection, and sink connection.

Data channels are communication paths between ECOs. Basically, a data channel is data unit that can be accessed by both source ECO and sink ECO. A data channel is uni-directional, which means the source ECO can only write data to the data channel, and its sink ECO can only read data from the data channel. To compensate the difference between the source ECO updating the data channel and the sink ECO consuming data, a data channel is designed as a buffer. The buffer size could be configured according to control application requirements. Table 5-5 shows the description of data channel.

Besides a data queue, each data channel also has a description to tell the underlying RTOS what ECOs it connects and what its other properties are. The data channel description specifies source and sink ECOs by their unique reference numbers and connecting port numbers. The data channel description also indicates manipulation methods. For example, when the data queue of a data channel is full and an ECO tries to write new data to the queue, the application can choose to overwrite the newest data in the data queue, overwrite the oldest data in the data queue, or block until space becomes available. The data queue overflow style can be configured in the data channel description, so that the data channel can be tailed to application requirements.

Table 5-5. Descriptor that defines a data channel.

```
typedef struct {
    Type_Tag           type;
    Node_Number        source;
    Port_Number        source_out_port;
    Node_Number        sink;
    Port_Number        sink_in_port;
    Overflow_Style     overflow_style;
    unsigned int       size;
    Array_Descriptor   array_dimensions;
    bool               interrupt_driven;
    void*              ISR_signal_value;
} DFG_Edge;
```

Table 5-5 shows the content of the data channel descriptor, which defines the channel's properties. In Table 5-5, "type" specifies the data type of the data channel, such as integer, float, or Boolean, etc; "source" and "source_out_port" indicate the source ECO and which output port the data channel is connected to; "sink" and "sink_in_port" indicate the destination for the channel's data; "overflow" indicates the behavior of the channel when the queue is full; "size" represents the size of the data queue; "array_dimensions" describes whether this data channel transmits multi-dimensional arrays or matrices of values, and if so, the dimensions of the matrix; "interrupt_driven" is a Boolean flag indicating whether the data channel's source is driven by an interrupt rather than by another ECO; for interrupt-driven data channels, the "ISR_signal_value" indicates which interrupt service routine (ISR) drives the data channel.

The basic data channel options are reading and writing data items. The read and write operations may cause input ports of the sink ECO or output ports to change status, and then cause the sink ECO or source ECO to change status. If a data channel is empty, a write operation will cause the inputs condition of the sink ECO changed. The firing rules of the sink ECO will be checked to see if any one is satisfied. Then this ECO process is ready to be executed. Which action will be taken depends on the waking firing rule. If an ECO tries to read from an empty data channel, this ECO process may blocked by the system until there is data available in the data channel. If an ECO tried to write to a full data channel, it will also be blocked by the system until there is new room in the data channel. In some extend, these data channel operations provides synchronization between source and sink ECO with different data updating frequencies.

5.4 Dataflow Graph

The dataflow graph describes the control software configuration as a composition of ECOs interconnected with data channels, as shown in Figure 5-8. Annotations on the graph specify ECO startup parameters, ECO priorities, ECO execution policies, data channel property choices, and data channel buffering policies. Designing a control application involves constructing such a dataflow graph by selecting ECOs from the design library and connecting them together. Additional user-defined or application-specific ECOs are also easily supported. ECOs within the dataflow graph can be allocated to different processors for distributed execution. Figure 5-9 shows two different partitioning decisions for allocating dataflow graph elements to two separate processor elements.

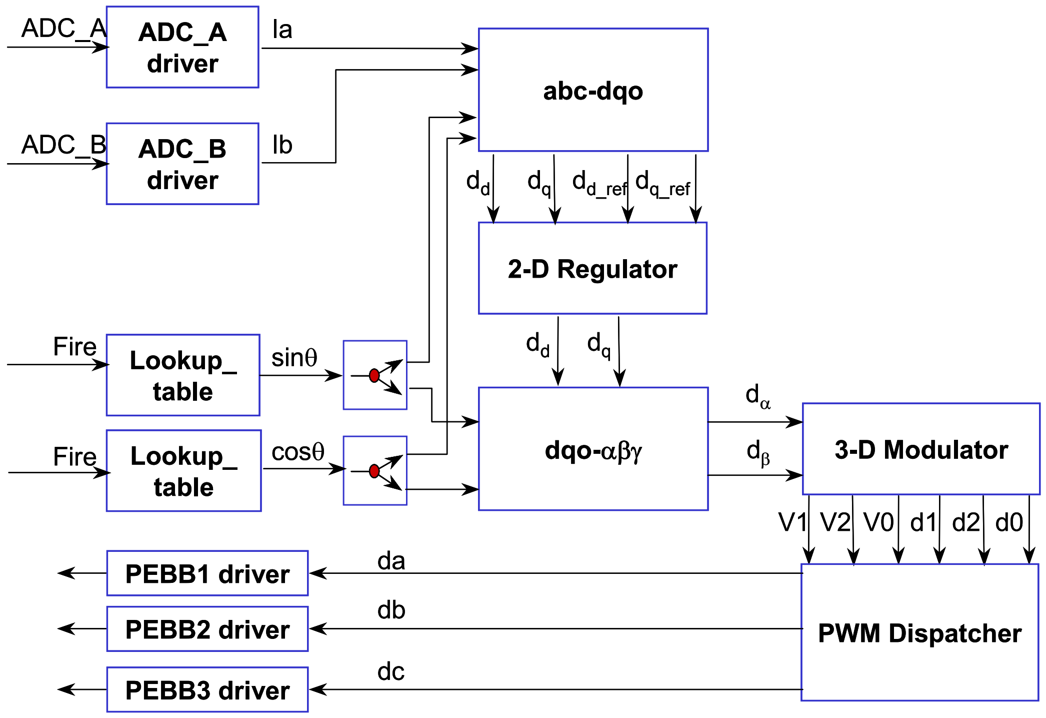


Figure 5-8. Dataflow graph of current close loop control for three-phase inverter.

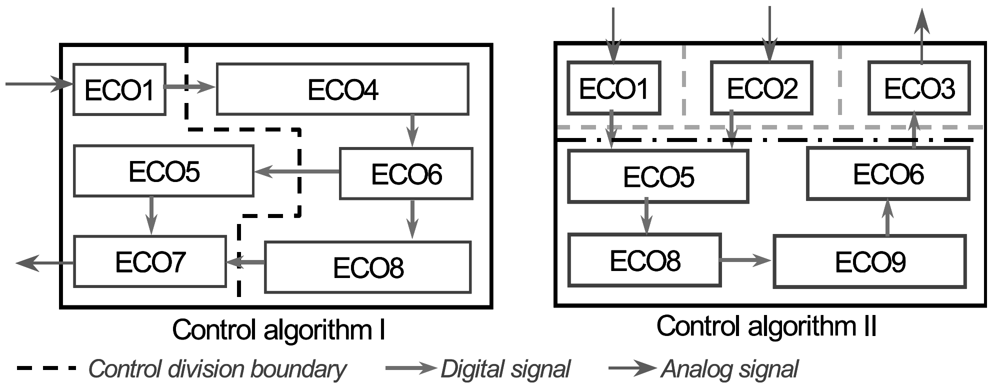
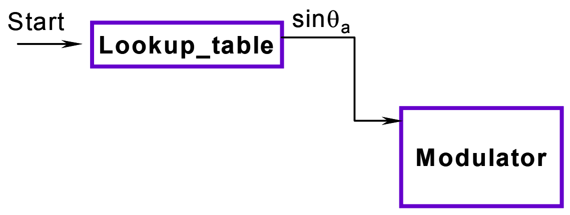


Figure 5-9. Dataflow graph examples.

Figure 5-10 shows an example of C code for a piece of dataflow graph in Figure 5-8, which contains two ECOs and two data channels.



(a) A piece of dataflow graph in Figure 5-8.

```
/*==========================================
 | Configuration Parameters
*-------------------------------------------------------------------*/
Lookup_table_Configuration Node_1_Configuration =
{ 672, 1, SIN_TABLE, 0, 0};

Modulator_Open_Configuration Node_4_Configuration =
{ 0.8, 50000};


/*===========================================*\
 | Node Set
\*===========================================*/
DFG_Node DFG_Node_Set[] = {…
  {&Lookup_table, &Node_1_Configuration,
   Lookup_table_Rule, 0, 7,
   LOOKUP_TABLE_FIRING_MASK_INCREMENT, 0},
  …
  {&Modulator_Open, &Node_4_Configuration
    , Modulator_Open_Rule, 0, 4
    ,    MODULATOR_OPEN_FIRING_MASK_DEFAULT,
0}
…}
```

(b) ECOs definition for (a).

```
/*===========================
 | Data Channels
*-----------------------------------------*/
DFG_Edge DFG_Edge_Set[] ={
{ Type_Bool
 , 0, 0      // Source name, port name.
 , 1, 0      // Sink name, port name.
 , OS_Block
 , 3         // Stores 5 elements.

 , 0         // Not an array.
 , true, 0 // Interrupt-driven.
 ,1          // updation frequency
 }
{Type_Float
,1, 0        // Source name, port name.
, 4, 0       // Sink name, port name.
, OS_Block
, 3          // Stores 5 elements.

, 0          // Not an array.
, false, 0 // Not interrupt-driven.
, 1          // updation frequency
}
```

(c) Data channels definition for (a).

Figure 5-10. An example of text-based dataflow graph.

5.5 The Dataflow Architecture Real-Time Kernel (DARK)

Unlike applications that are built up from simple subroutines, dataflow applications require support for their unique features, including support for concurrent execution, data channel buffering, interprocess synchronization and interrupt handling. One approach to providing this infrastructure is to encapsulate it in a small, embeddable, real-time operating system (RTOS). Such RTOSs are often called "micro-kernels" because, in comparison to full-featured RTOSs, they are stripped of all but the most minimal features in order to provide extremely efficient services within a minimal memory footprint. For these purposes, a custom real-time micro-kernel was designed. The dataflow real-time kernel (DARK) [105] should support lightweight process management, data channel management, system resource allocation, device driver support, and interrupt handling.

For control of single converter systems, the real-time computing is carried with a single rate, which is the switching frequency. The execution sequence and rate of each ECO process can be predefined. Hence, for single rate control applications, static scheduling will be sufficient, and no data buffering is necessary. In the case of systems where multiple converters coexist, it is possible that ECO processes are executed at different rates. This requires DARK to provide dynamic scheduling mechanism and data buffering. In order to handle emergency or aperiodic events, preemptive scheduling would be necessary. Thus, DARK is designed to be scalable for different situations:

- Preemptive multi-thread ECO scheduling with queue data channels
- Preemptive multi-thread ECO scheduling with mailbox data channels
- Non-preemptive multi-thread ECO scheduling with queue data channels
- Non-preemptive multi-thread ECO scheduling with mailbox data channels
- Single-thread dynamic ECO scheduling with queue data channels
- Single-thread dynamic ECO scheduling with mailbox data channels
- Single-thread static ECO scheduling with queue data channels
- Single-thread static ECO scheduling with mailbox data channels

Among those DARK options, preemptive multi-thread ECO scheduling with queue data channels involves the maximum overhead, and Single-thread static ECO scheduling with mailbox data channels presents the least footprint.

5.6 ECO Library

It is also mentioned in 3.5 that the open control architecture contains a design library, which provides control designer standard large piece of functions. Then the problem is what functions should be included as an ECO. As discussed in 5.1, the granularity and efficiency of the code are tradeoffs. The more granular the ECOs, there needs more number of ECOs to implement the same control algorithm, which also means more system running cost on interactions between modules. On the other hand, to achieve higher code efficiency means reducing the granularity of ECOs. If we use the number of modules (N) in a dataflow graph to leverage the granularity, and use the ratio η of execution time for computation and ECOs interactions, which is:

$$\eta = \frac{execution\ time\ for\ computation}{total\ execution\ time\ of\ one\ round\ of\ dataflow\ graph}$$

For example, to implement a simple open-loop SPWM control for a 3-phase VSI shown in, there might be different choice of ECOs.

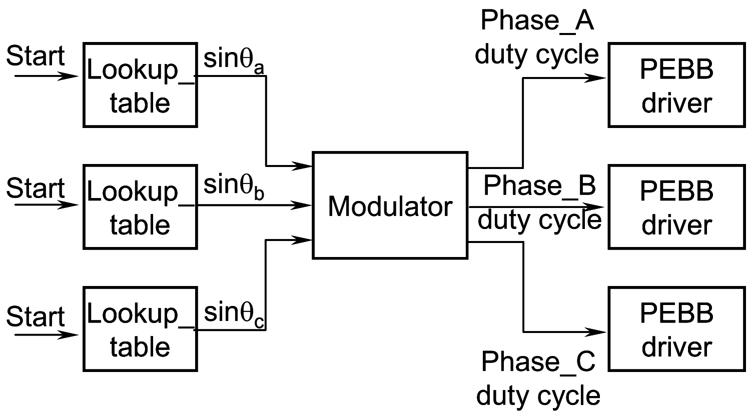


Figure 5-11. Open-loop, three-phase inverter.

ECO Group 1: +, -, * , /, sin(), cos();

ECO Group 2: ECOs as shown in Figure 5-11;

ECO Group 3: the whole open-loop control integrated as one ECO.

If the least feature of DARK is chose, which is single thread static scheduling with mailbox method for data channel oprations, the exection time for basic ECO interactions are shown in Table 5-6. Then, the number of ECOs need to be used and code efficiency can be estimated using ECO groups with different level of granularity, as shown in Table 5-7 through Table 5-9.

Table 5-6. Execution time of basic DARK operations.

Operations	*Number ofDSPInstruction Cycles*
Data channel read	8
Data channel write	12
ECO switching	20

Table 5-7. Estimate of code efficiency ECO group 1.

ECOs	*Computation*	*ECO Interaction*
+	1	40
-	1	40
*	1	40
/	1	40
sin()	20	40
cos()	20	40

Table 5-8. Estimate of code efficiency ECO group 2.

ECOs	*Computation*	*ECO Interaction*
Look_up_table	20	40
Modulator	40	80
PEBB_driver	30	40

Table 5-9. Estimate of code efficiency ECO group3.

ECOs	*Computation*	*ECO Interaction*
Open loop	200	90

Table 5-10. Estimate of code efficiency.

ECOs grougs	***N***	η
Group 1	50	0.08
Group 2	7	0.55
Group 3	1	0.9

It shows Group 2 has reasonable number of nodes and code efficiency. Also dataflow graph using Group 2 naturally represent the control algorithms. On the other hand, when the system running resource is very limited, Group 3 could also be a choice.

Chapter 6: ASSESSMENT AND VERIFICATION OF DATAFLOW SOFTWARE ARCHITECTURE

In this chapter, the open control architecture implemented using the dataflow approach will be assessed regarding feasibility as well as complexity of software development. Four control applications with different complexity will be used to compare dataflow-based software with a corresponding monolithic C version. Furthermore, to verify the dataflow control architecture, the dataflow-based control software for a 3-phase VSI will be implemented on a PEBB-based hardware system.

6.1 Dataflow Architecture Assessment

The reason that dataflow was chosen as the implementation style for the proposed open control architecture is the reduced complexity of control software composition and maintenance, the flexibility to transplant to different hardware platforms and the ease of adapting to adapt to distributed computation environment. While the software engineering benefits of using a compositional, component-based architectural design are well known [90], the perceived overhead imposed by dataflow may become a serious obstacle to the feasibility of dataflow software in the real-time control application. The dataflow architecture for power conversion system control is assessed regarding these two folds.

6.1.1 Dataflow Applications

We implemented the embedded control software in several of applications using a dataflow architecture. The open-loop, three-phase inverter is the simplest application, while the four-leg inverter is a fairly complicated application.

6.1.1.1 Open-Loop Three-Phase Inverter

The dataflow graph (DFG) for the open-loop, three-phase inverter application is shown in Figure 6-1. This is the simplest application used in our experiments. The control algorithm is sinusoidal PWM (SPWM). It consists of three *Lookup_Sin* ECOs that receive a *Start* signal from their Boolean input data channels. They look up a value from a circular table that they maintain using a table pointer. After every look-up, the table pointer is incrementally increased. The table source and the modification step for the table pointer are stored as part of the ECO's configuration information. The output values of the three ECOs have a phase difference of 120 degrees. In combination, these form input for the *Modulator* and activate it to produce three floating-point results that form inputs to the three *PEBB_drivers*. The *PEBB_drivers* convert the data from the floating-point format to a format of control information that can be understood by the power stage. In the following two applications, the inputs are relatively more sophisticated than those of the open-loop application.

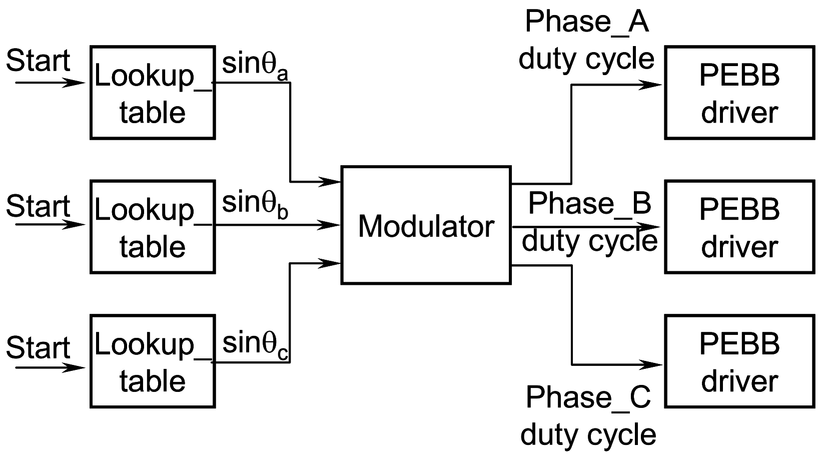


Figure 6-1. Open-loop, three-phase inverter.

6.1.1.2 Boost Rectifier

Figure 6-2 shows a dataflow graph for a three-phase boost rectifier with closed-loop control. There are two control loops: the current loop and the voltage loop. The voltage loop should be executed first in order to generate a reference for the current loop. For the three-phase current loop control, the dq transformation technique is used.

All sensed data are implemented using interrupt-driven data channels, and are synchronized by the switching clock. The rising edge of the switching clock causes the execution of the corresponding interrupt handler, in which all the sensed data, phase currents and voltages, and DC voltage are updated. The ADC drivers then translate those sensed data to their correct values.. At this point, the ECO *1-D regulator* and *synchronize* commands ready. The DC voltage is regulated in the ECO *1-D regulator*, and the current loop reference d_{d_ref} is generated. The *synchronize* ECO tests the phase voltages and generates a Boolean output to indicate whether or not the phase angles need to be synchronized.

The two *lookup_table* ECOs have two different behaviors depending on different inputs' combinations. At the normal condition, if the phase voltages do not need synchronizing, the *lookup_table* ECOs incrementally increase their table pointers and output the table value; otherwise, the table pointers will be reset. Through the ECO *duplicator*, the sin and cos values are copied and directed to two different ECOs. So far, the *abc_dqo* ECO is ready to transform the phase currents in abc coordinates to dqo coordinates. Then the *2-D regulator* performs the current loop regulation using the reference generated from the voltage loop. The regulated currents in dqo coordinates will then be transformed back into αβγ coordinates through ECO *dqo-αβγ*. Then, the ECO *3-D Modulator* is ready to synthesize duty cycle information for each phase. In the *PEBB_driver* ECO, the duty cycle information will be translated to the form that can be used to generate a switch pulse at the phase leg.

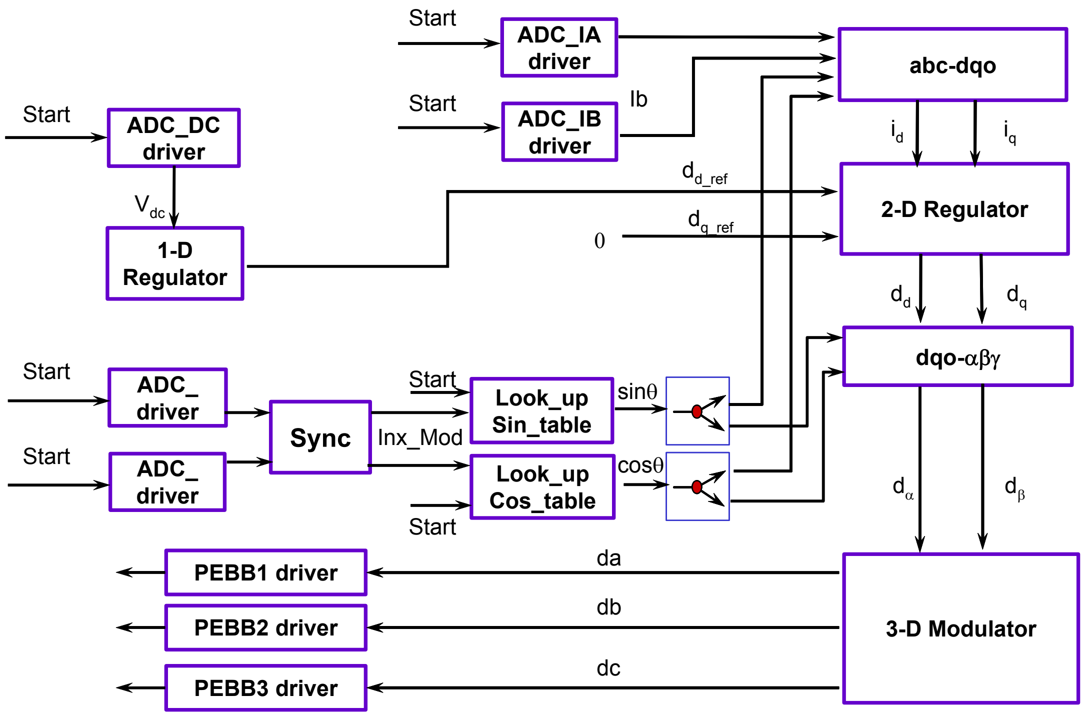


Figure 6-2. Closed loop control for three-phase boost rectifier.

6.1.1.3 Closed-Loop Three-Phase Inverter

Figure 6-3 shows an ECO-based application for closed current loop control of a PEBB-based three-phase inverter. The dq transformation technique is used to simplify the control. At the beginning of a switching period, the driver ECOs for A/D converters are fired to read feedback information (phase currents) from sensors; the two *Lookup_table* ECOs are fired to output sin and cos data. Since the same sin and cos data are used more than once in the dataflow graph, a *1-to-2 Duplicator* ECO? is used to send the same data to two places. "In the *abc-dqo* ECO, the phase currents in abc coordinates are transformed into dqo coordinates. The *2-D Regulator* ECO performs PI regulation in dqo coordinates. The *dqo_αβγ* ECO transforms the duty cycles back into αβγ. Then the *3-D Modulator* performs modulation on the duty cycles. The *3-D Modulator* generates the duty cycle information for the whole power stage. Depending on the physical distribution of the power stage, the *PWM Dispatcher* synthesizes the duty cycle information for each separate hardware asset, in this case, phase-leg PEBBs. Each PEBB driver will then translate the floating-point format duty cycle into the proper format for the driving circuit on the PEBB.

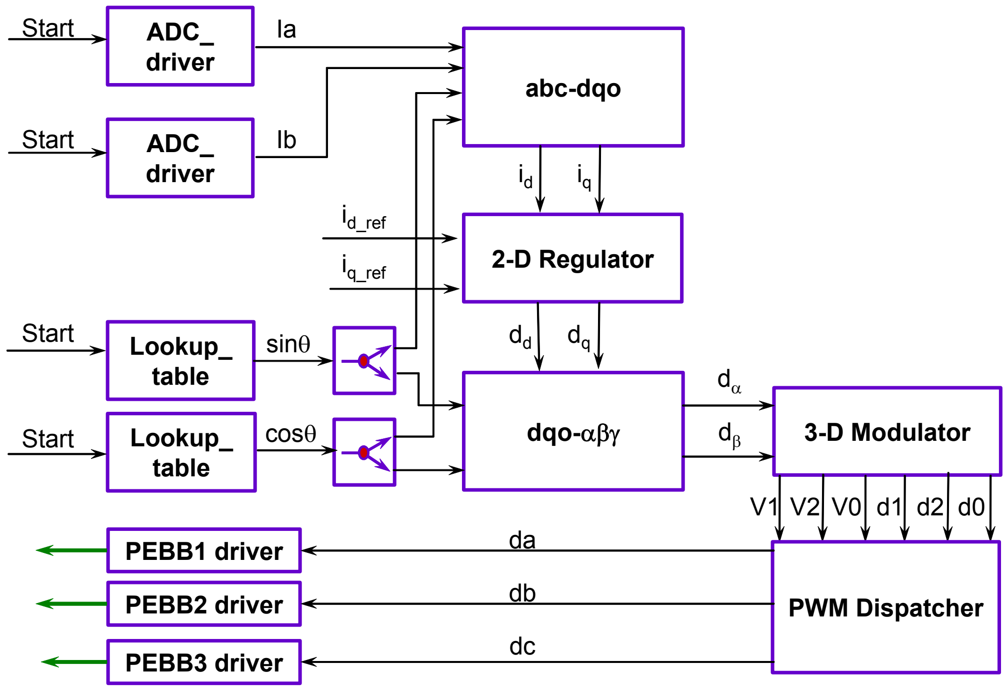


Figure 6-3. Closed-loop three-phase inverter.

6.1.1.4 Closed-Loop Four-Leg Inverter

Figure 6-4 shows an ECO-based application for closed voltage loop control algorithm of a PEBB-based, four-leg inverter. The dq transformation and SVM techniques are still used, but different ECOs will be used because the dq transformation and SVM in the four-leg inverter have one more dimension than those in the three-phase inverter control application. Since in this application, the four phase voltages are the feedback information, different *ADC driver* ECOs are used for the A/D converters on the four PEBBs. The *Lookup_table* ECO is reused. If the PEBBs are the same as in the previous application, the *PEBB driver* ECOs can also be reused. One more *PEBB driver* ECO is needed for the fourth leg.

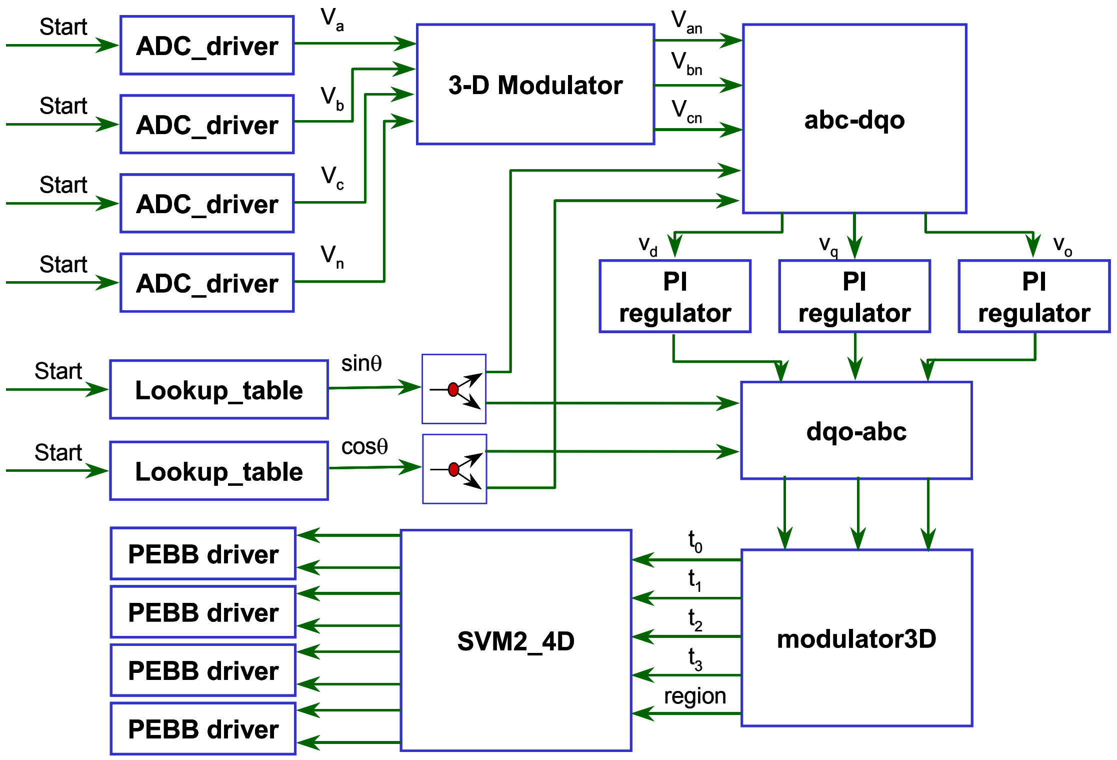


Figure 6-4. Dataflow graph of voltage close loop control for 4-leg inverter.

6.1.2 Feasibility

Given these concerns, the question then arises: is it feasible to use dataflow for power electronics control applications, or are these costs too prohibitive?

In order to explore the feasibility of applying the dataflow approach to real-time power electronics control tasks, a preliminary study of the performance issues was undertaken. A typical control application – closed-loop current control for a 3-phase PEBB-based inverter as shown in Figure 6-5 – is implemented under dataflow architecture with different DARK options. To assess the feasibility of the dataflow approach, the code performances of dataflow software will be compared to those from the legacy custom-designed C code and from the Real-time Workshop Embedded Coder [106]. All implementations were written in C.

The specifications of the three-phase VSI are:

Input: Vdc = 200 V;

Outputs: balanced three-phase sinusoidal with line-to-line voltage of 200 V;

Switching frequency: fs = 10 KHz;

For each phase:

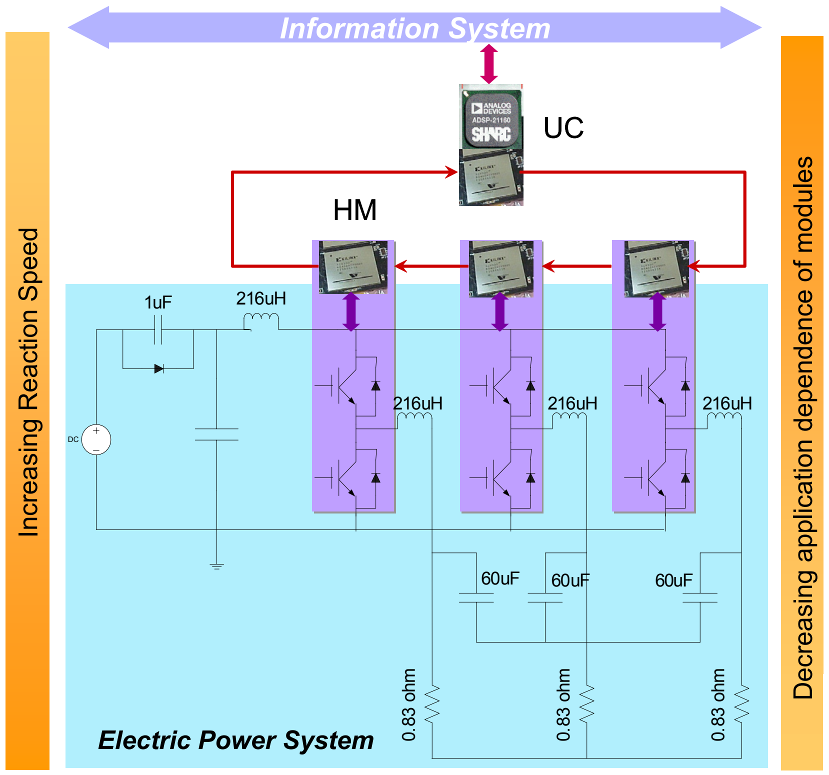
Switches: 1200 V, 300 A HB IPM

Output inductance L = 216 uH at each phase;

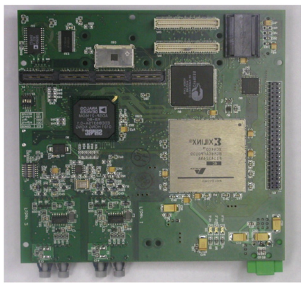
Output capacitance C = 60 uF at each phase.

Load resistance R = 0.83 ohm.

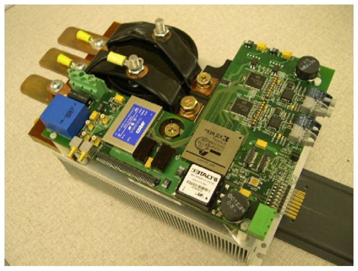
The current loop is designed to have a phase margin of 35 degrees and 10dB gain margin. The dataflow graph of the embedded control for the three-phase inverter is shown in Figure 6-3.



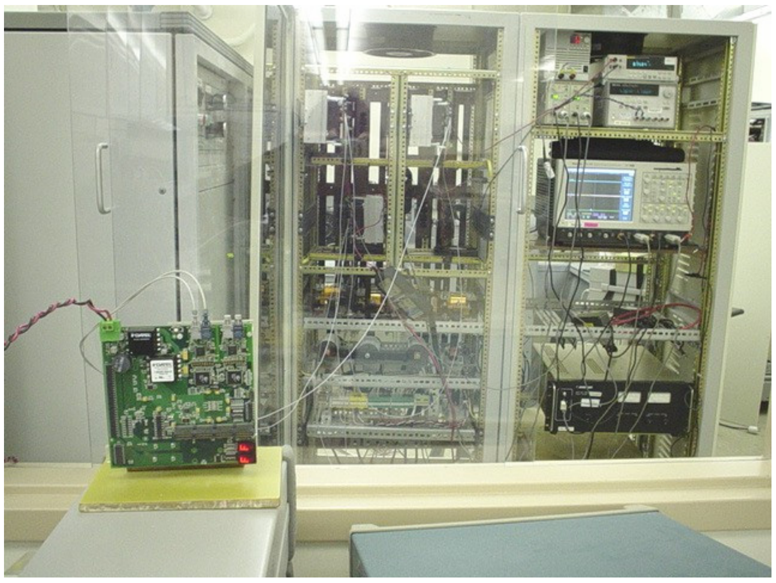
(a) Block diagram of the system.



(b) Universal Controller.



(c) PEBB module.



(d) Test setup.

Figure 6-5. A PEBB-based power converter.

The code performance is measured by the actual execution time of one round of the complete control loop for different versions of dataflow architecture running on the same Analog Device 80MHz SHARC DSP (ADSP) 21160. For multi-thread scheduling, normally people refer this kind of response time as sojourn time. With the help of profiling tool in software development environment provided by Analog Device, it is possible to count the number of DSP instruction cycles used for the execution of specified trunk of software. One instruction cycle of ADSP is 1/80 MHz, i.e. 12.5 ns. 21160 Thus, the actual execution time of one round of the control loop can be estimated fairly accurately as (number of instruction cycles * 12.5 ns).

Figure 6-6 shows the performance comparison between the three versions of the three-phase closed-loop control inverter application. For the dataflow architecture, the performances of the same dataflow graph with various DARK options is shown. The measurement of execution time is also categorized by different operations. From Figure 6-6, the additional overhead imposed by the use of concurrent processes is evident. The instruction cycles devoted to "data channel operations" were those executed by the data channel read and write system calls made by ECOs in the dataflow design. The "ECO scheduling" instruction cycles were those devoted to determining which ECO was to be executed next. The "context switching" instruction cycles were those devoted to saving or restoring register contents when switching from the currently active process to the RTOS, or when switching from one active process to another.

The DARK used for this comparison employed streamlined choices for how to manage ECO processes in order to provide a picture of what may be feasible. It supported ECOs by providing non-preemptive scheduling and by supporting single-unit data channels (instead of multi-item queues). More comprehensive RTOS features, including preemptive task switching, full support for dynamic process priorities, and arbitrarily sized data channels, are all possible. The "mailbox" messaging method provides faster connections between ECOs, but with less protection against data loss due to different update frequencies of source and sink ECOs. The "queue" messaging method, on the other hand, is slower at passing data but provides a buffer between ECOs with different updating frequencies. When the ECO execution sequence can be predicted, static scheduling introduces the least performance overhead for context switching and process scheduling. If there are control events occurring at different frequencies, dynamic scheduling allows for more flexibility in control. The multi-thread allows better concurrent control in the system, and pre-emptive scheduling makes the control software capable of responding to the most critical events in the order of highest priority. These features will each introduce additional run-time overhead, however.

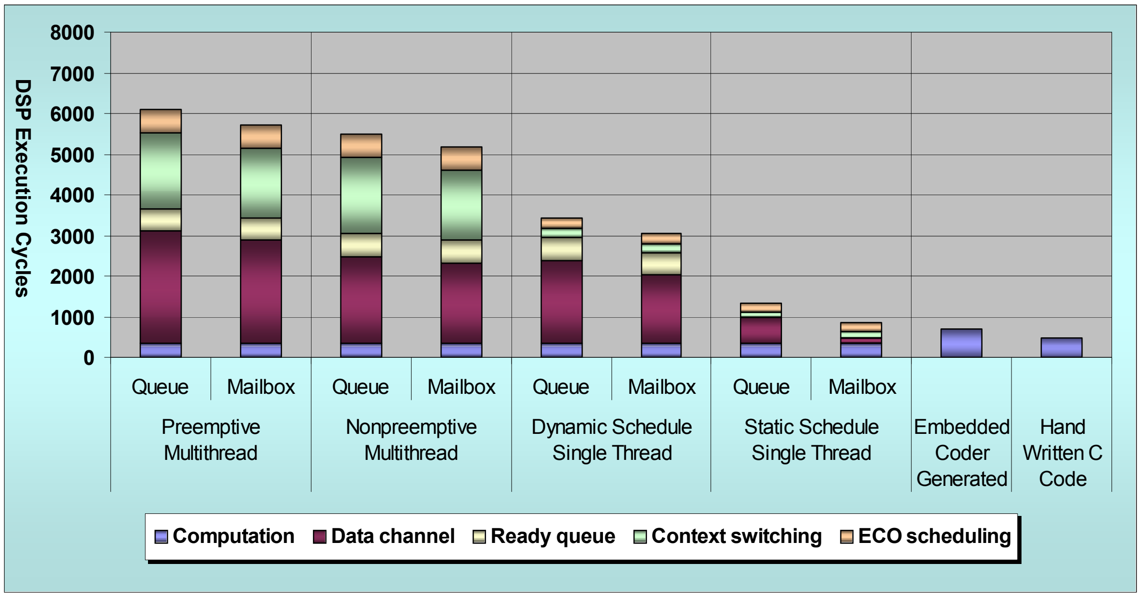


Figure 6-6. Code performance comparison.

For the closed-loop, three-phase inverter control application, the DARK with the smallest footprint – mailbox and static single thread – is sufficient to provide full control to the power processing system. Figure 6-6 shows that with this DARK feature, the performance of the dataflow software can be compared to the C code generated from the Real-time Workshop Embedded C Coder, which is also adopted by many industry control software platforms, such as ABB OPCoDe.

From Figure 6-6, it is also clear that inter-process communication, in the form of data channel actions, dominated the overhead introduced by the ECO approach. Context switching and process scheduling were also important factors in the increased time required by the ECO application. Nevertheless, the application still ran within the limits necessary for proper performance.

Also, it is important to note that this experiment does not provide an accurate picture of the relative fraction of time spent on overhead issues versus the time spent in computation. That is because the computational aspects of this control algorithm are so small as to be negligible. This provides a much better picture of the absolute amount of overhead introduced, but does little to give a true impression of how this compares to a realistic computational burden.

6.1.3 Engineering Effort Endeavored to Control Software Development

Under the main-program-and-subroutine style, because the software is close coupling to hardware, almost every new application needs to develop from the scratch. Under the dataflow-based open architecture, the control software development is naturally component-based and mainly is to express the control with its corresponding dataflow graph. The written code size in terms of characters is also measured to show to some extent the impact of software architecture and style on the complexity of software development.

Figure 6-7 shows the rewritten code sizes under main-program-and-subroutine and dataflow style, respectively, for the four applications presented previously in this chapter. For the main-program-and-subroutine style control software development, the written code size is roughly the size of the control code itself; while for dataflow style software, the rewritten code size in text-based development environment will be the size of text used to describe an application specific dataflow graph. Since code size is a factor of software complexity, Figure 6-7 also shows the larger the code size, the more reduced rewritten effort involved in the dataflow software development compared to main-program-and-subroutine style software. It is worth to mention that code size is not the only measurement of software complexity, the context is also important. For dataflow style, the text-based software development is specifying the selection of ECOs and connection of ECOs. For main-program-and-subroutine style software, the software development still includes math or logic manipulation as well as hardware interfaces.

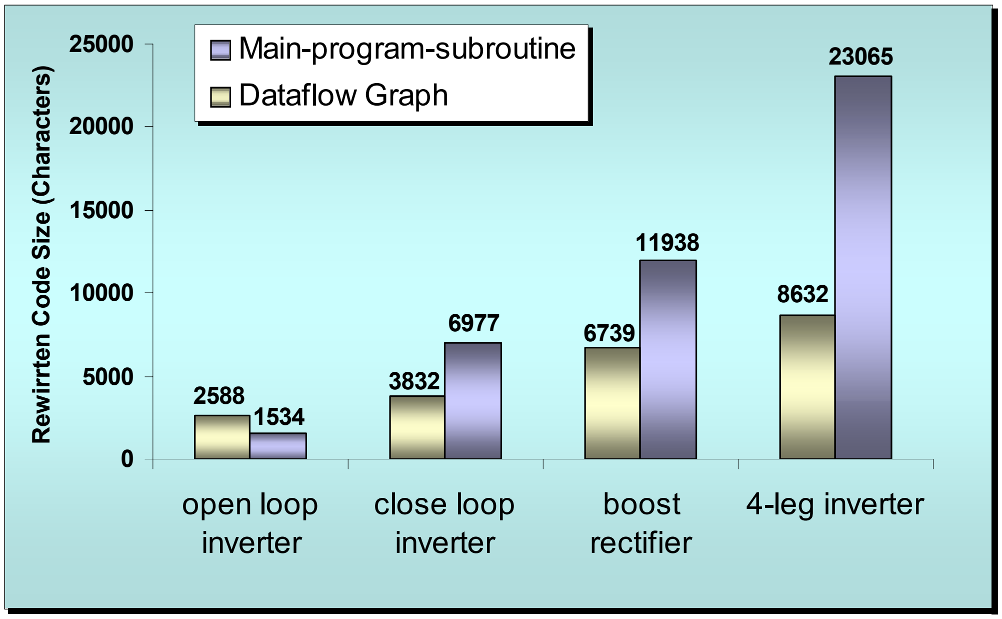


Figure 6-7. Comparison of code size.

6.2 Dataflow Architecture Verified on PEBB-base Converter Hardware System

After the preliminary assessment of dataflow architecture from a software point of view, this approach is also verified on a PEBB-based 3-phase inverter system, which aims at modularization and standardization of power hardware.

6.2.1 PEBB-based Converter System

The PEBB-based converter system designed at CPES is shown in Figure 6-5. The power hardware system is composed of PEBBs and passive components. The control system is composed of Universal controller (UC) and hardware managers (HM). UC conducts converter level control, HM is integrated with PEBB for local intelligence such as gate signal synthesis and local protection. UC and HMs communicate through a so-called PESNet (power electronics system network), which is actually an optic fiber ring.

The UC has been designed to address the needs of the majority of power electronics applications in medium to large power electronics converters. Large systems need communication interfaces, status indicators, debugging tools, advanced control algorithms and fault tolerance. Figure 6-8 [107] shows a function block diagram of UC. The CPU is composed of an SHARC ADSP 21160 and Xilinx FPGA XC18V04-VQ44C. The DSP is taking most of the computation of control and the FPGA provides support to various peripherals. All the control software written in C is finally complied into machine code is downloaded and running in the DSP.

The role of HMs is actually interfaces between the control and the power stage. On one side, the HM connects into the information system via optical fibers arranged in single or dual ring structure. On the other side, the HM connects to the power stage through a phase leg, forming a basic PEBB. Figure 6-9 [107] shows the function block diagram of an HM. The core of HM is a Xilinx FPGA, which collaborates gate drives, ADCs and communication to higher level control.

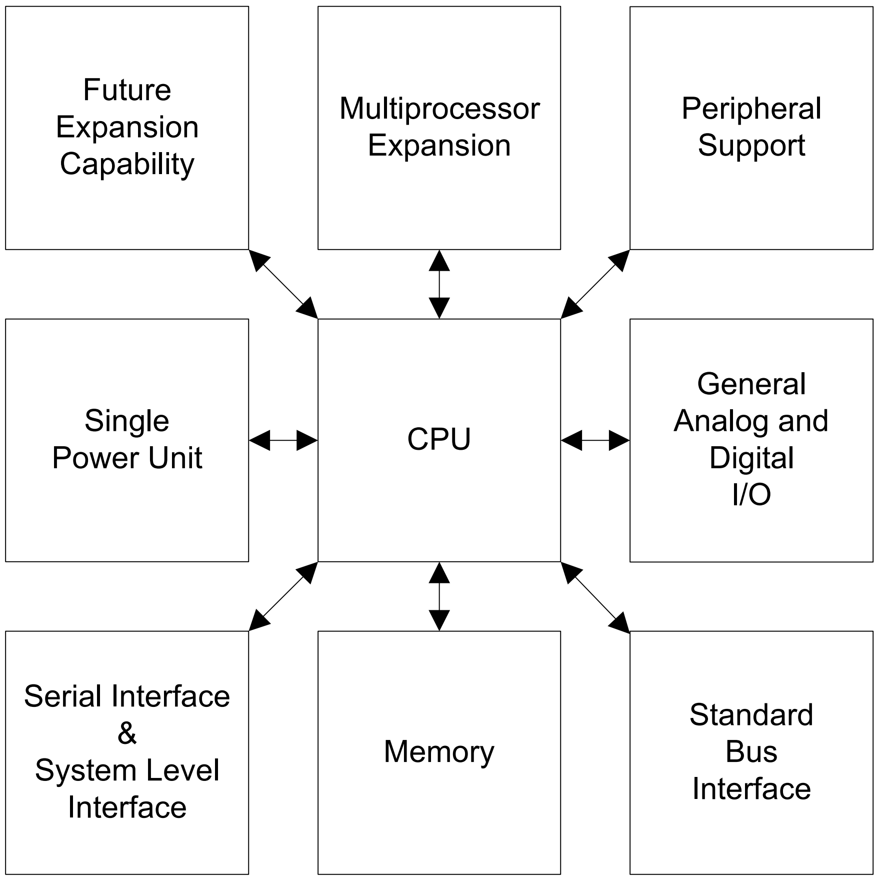

Future Expansion Capability
Multiprocessor Expansion
Peripheral Support
Single Power Unit
CPU
General Analog and Digital I/O
Serial Interface & System Level Interface
Memory
Standard Bus Interface

Figure 6-8. Function block diagram of UC.

In such a PEBB-based power converter system, the massive computation related to converter control is carried by UC. The switch control commands are sent to each HM, and will be interpreted into gate signals by HM. For closed-loop control, HM will also be responsible for sending sensing data acquired at local sensors back to UC. The key of connection between UC and HMs is the so-called PESNet.

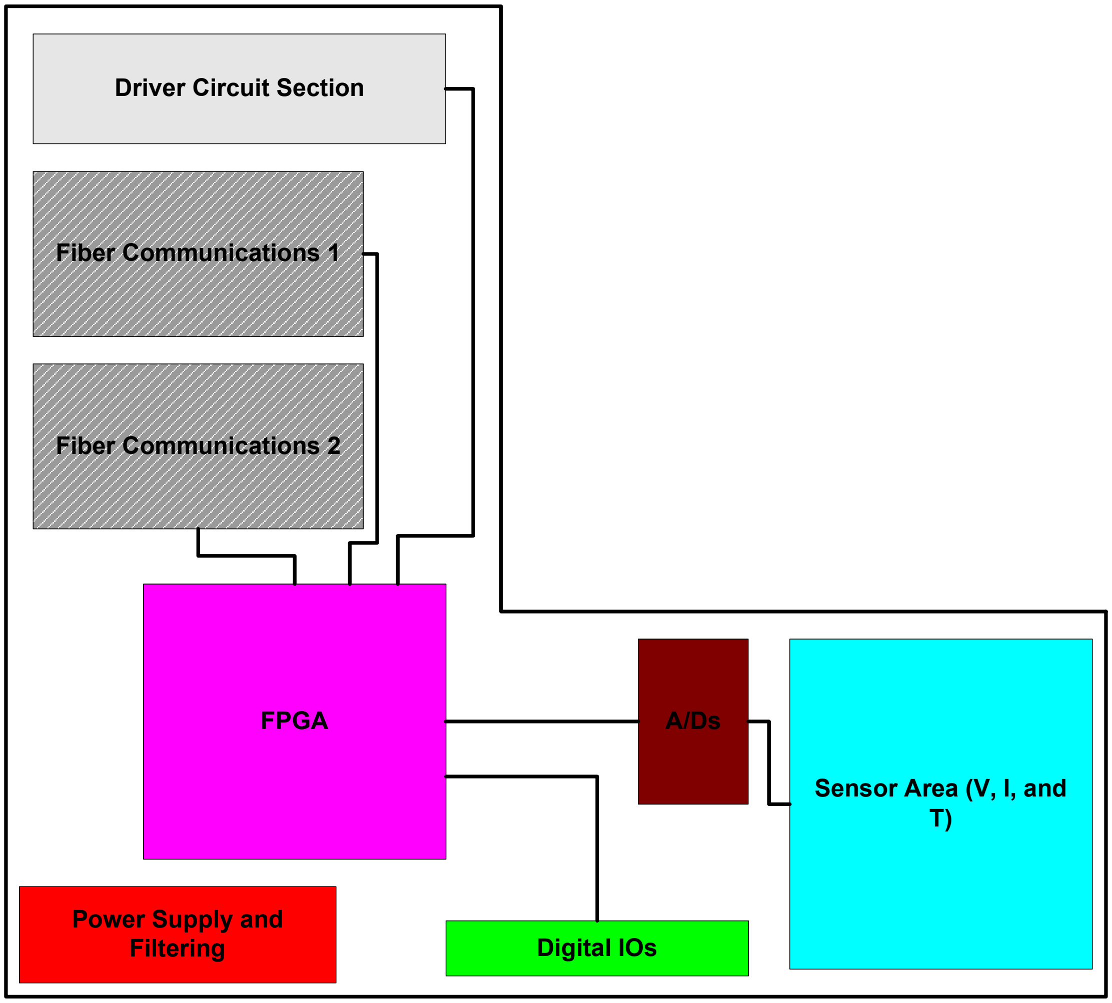


Figure 6-9. Function block diagram of HM.

6.2.2 Power Electronics System Network II (PESNet II)

In PEBB systems, control architecture tends to be distributed, which requires communication networks. Some features of power electronics systems, such as hard real-time, life critical, etc., impose intense considerations in design digital communication networks, which should provide efficient information interchange and system protection mechanism. This will address several important issues in design of digital communications in power electronics systems. First, to provide fast communication and support flexible system construction, the candidate network topologies will be explored. Meanwhile, the EMI immunity is taken as an important consideration to design the network topology, since the communication is most likely carrying on in a rather noisy environment. To address these issues, a fiber optic ring structure is selected, as shown in Figure 6-10, for the digital communication in PEBB-based system.

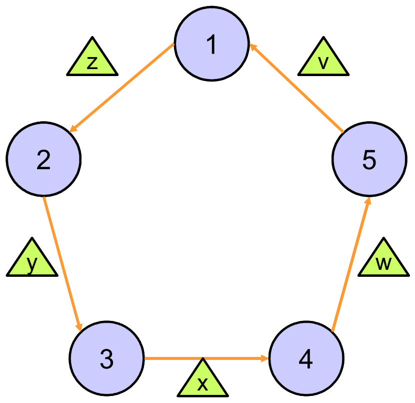


Figure 6-10. Optical fiber ring structure.

6.2.2.1 PESNet II Structure

The so-called PESNet II protocol has been design for the optical fiber ring to support fast communication between UCs and HMs and closed-loop control capability. The protocol stack contains four layers: physical layer, link layer, network layer and application layer. The physical layer includes the optical fibers and TAXI chipset used as transmitter and receiver. The link layer and network layer are implemented in the FPGA of both UCs and HMs, while application layer is implemented in the DSP of UC and serves as interface for control software to connect into the information network in PEBB system.

At the network layer, the PESNet support peer-to-peer communication. In PESNet II, the network address for a node is dynamically assigned by UC. The packet format is shown in Figure 6-12. The packet header CMD is a 4-bit field that indicates the type of a packet. Five basic types of packets are defined. Among them, NORMAL packets are specially designed for switch turn on/off control and exchange of sensing data for closed-loop control. EXTENDED packets are used for network configuration. SRCADDR is an 8-bit field specifying the sender of the packet, and DSTADDR is also 8-bit for the receiver of the packet. NETTIME is an 8-bit field used for nodes synchronization, which will be discussed in more detail later. DATA is the data payload of a packet with configurable length. CRC is an 8-bit field for fault diction and correction.

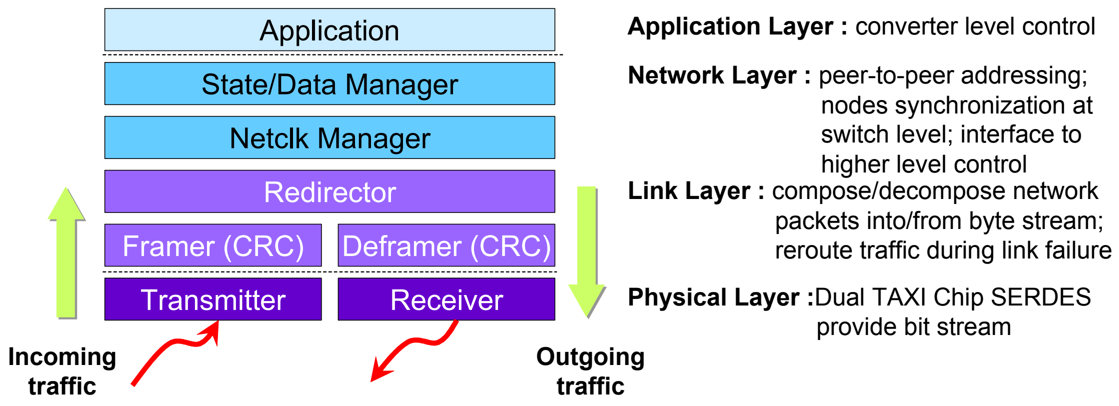

Application
State/Data Manager
Netclk Manager
Redirector
Framer (CRC)
Deframer (CRC)
Transmitter
Receiver
Incoming traffic
Outgoing traffic
Application Layer : converter level control
Network Layer : peer-to-peer addressing; nodes synchronization at switch level; interface to higher level control
Link Layer : compose/decompose network packets into/from byte stream; reroute traffic during link failure
Physical Layer :Dual TAXI Chip SERDES provide bit stream

Figure 6-11. PESNet II protocol stack.

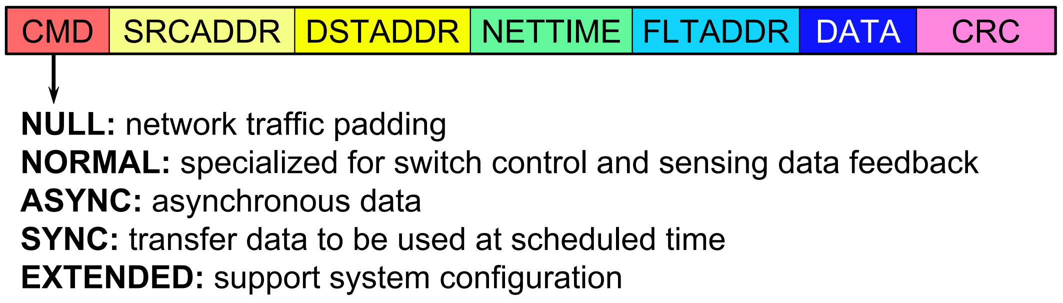

CMD
SRCADDR
DSTADDR
NETTIME
FLTADDR
DATA
CRC
NULL: network traffic padding
NORMAL: specialized for switch control and sensing data feedback
ASYNC: asynchronous data
SYNC: transfer data to be used at scheduled time
EXTENDED: support system configuration

Figure 6-12. Data packet format.

6.2.2.2 Nodes Synchronization Mechanism

Restrict synchronization are necessary between PEBBs in voltage-fed converter systems. However, for the ring structure, there is no natural synchronization between network nodes. A network clock mechanism is designed so that all the nodes in the network can be synchronized at us level with around 80ns jitter, which should be sufficient for switching power applications up to 50kHz. The synchronization is illustrated in Figure 6-13.

For every node, a packet is received and sent out per 2us. Even if a node does not have data packet to send out, it will pad the network with a NULL packet. Every node keeps a so-called network clock (NETCLK) which is synchronized to the packet sending. So every 2us when a packet sent out, the NETCLK should also get incremented. Every node also maintains a packet receiving clock, which is started by the arrival of a packet at the transmitter. From the sending of a packet at one node until the sending of a packet at the next node, the time elapsed is set to be 2us. By pre-adjust the delay between packet sending clock and packet receiving clock, it is possible to adjust all the nodes to send packet at roughly the same time. The jitter is caused by a higher resolution bit stream clock of TAXI chipset, and is around 80ns.

If NETCLK is set to be increment with the packet sending, the increment action of NETCLK at different nodes can be synchronized. The problem left is how to adjust the value of NETCLK at every node, so that they have the same clock value. For this purpose, one and only one node in the network is selected as a NETCLK master, which starts it NETCLK with an arbitrary value, and then increment NETCLK every time it sends out a packet. For the rest nodes in the network, they will adjust its own NETCLK to the value in a packet sent from the NETCLK master. Thus, all the nodes can have the same NETCLK value. For operations taken by different nodes but need to be synchronized, the sender of these operations specifies the schedule net time for these operations to be fired. When a node receives such a packet for a time stamped operation, it will fire the operation when its own NETCLK reaches the schedule time. The center aligned gate signals for top switch at 3-phase PEBB converter system is shown in Figure 6-14.

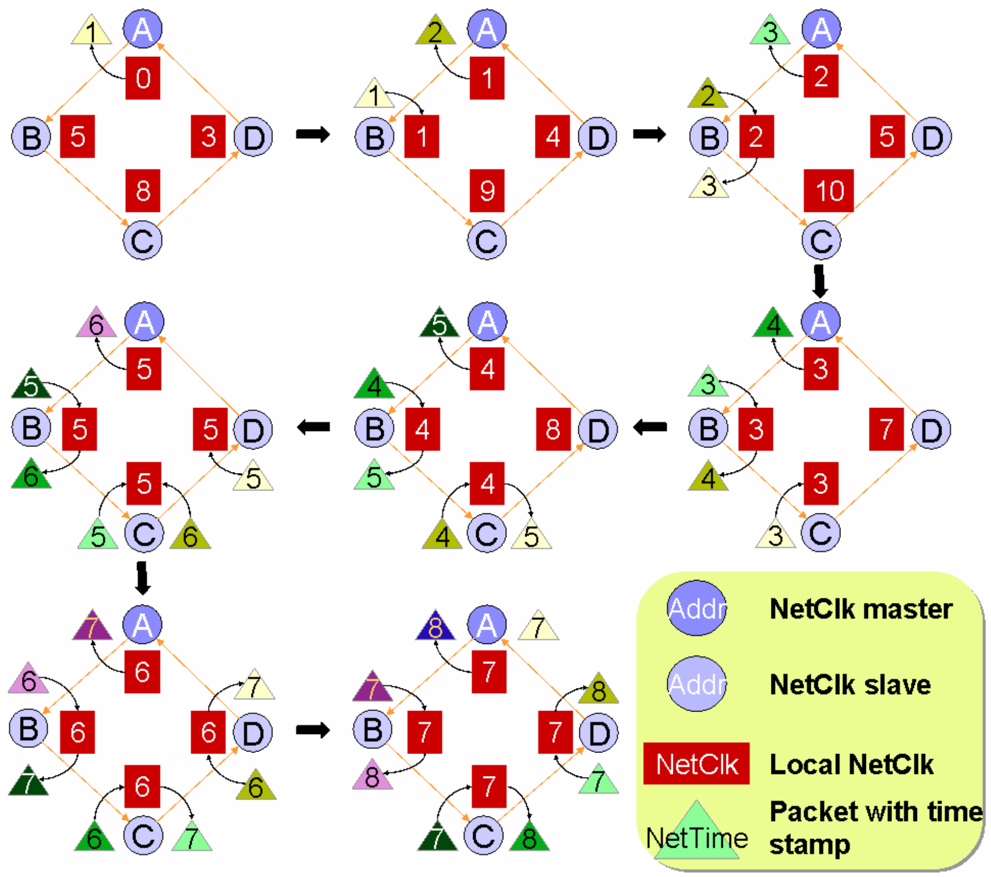


Figure 6-13. Synchronization mechanism in PESNet II.

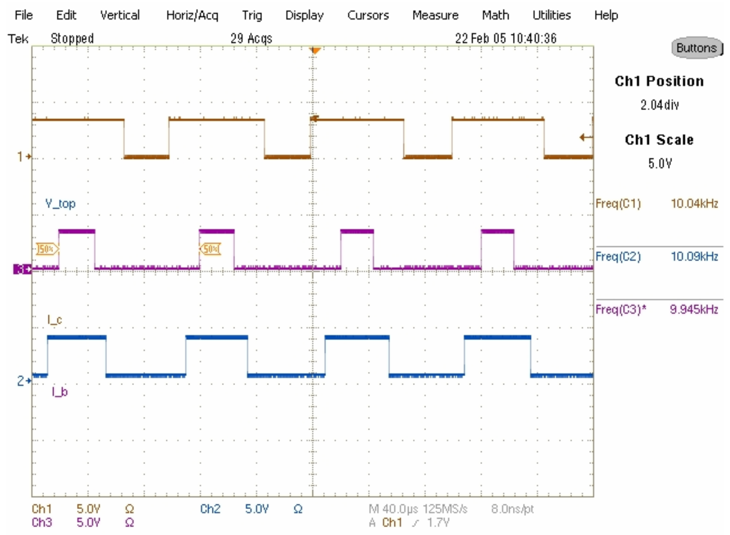

File Edit Vertical Horiz/Acq Trig Display Cursors Measure Math Utilities Help
Tek Stopped 29 Acqs 22 Feb 05 10:40:36
Buttons
Ch1 Position 2.04div
Ch1 Scale 5.0V
Freq(C1) 10.04kHz
Freq(C2) 10.09kHz
Freq(C3)* 9.945kHz
V_top
I_c
I_b
Ch1 5.0V Ω Ch2 5.0V Ω M 40.0μs 125MS/s 8.0ns/pt
Ch3 5.0V Ω A Ch1 ∕ 1.7V

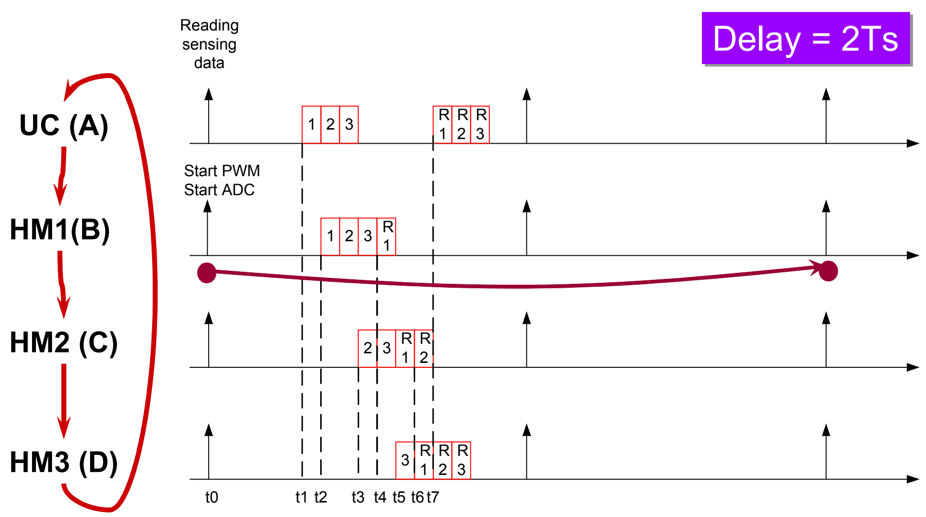
Figure 6-14 Synchronized center aligned gate signals.

6.2.2.3 Latency

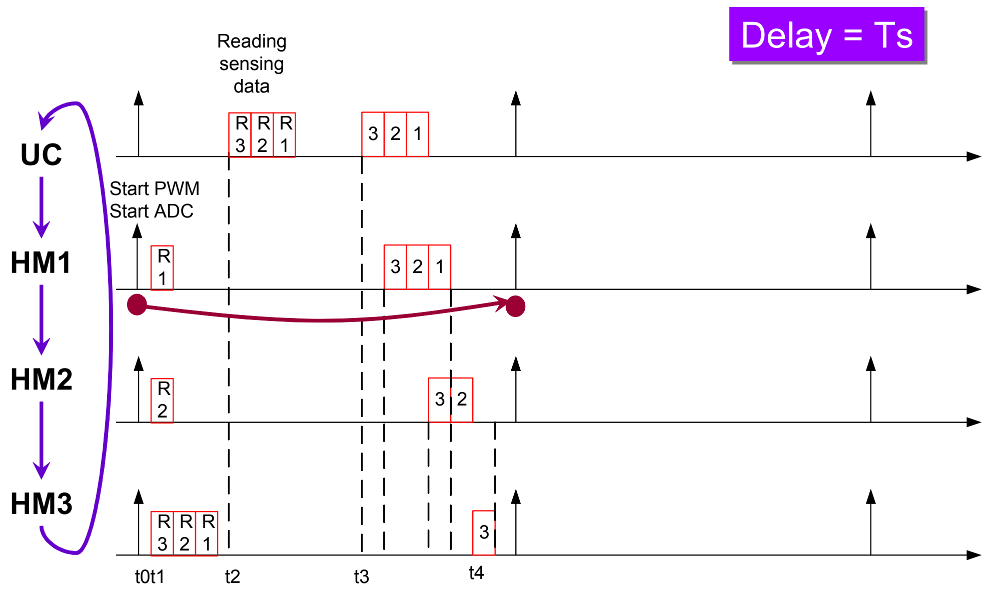
Due to the limited bandwidth of communication network and all the digital components signal processing speed, there will be digital delays associated with the PEBB system, as shown in Figure 6-15. Digital delay could detrimental overall control performance as will be discussed later.

Different digital delays may occur due to the sequence of handshaking between UC and HMs. For an open loop control, the digital delay is defined as the real time elapsed from a switch command computes until a switch begins to act. In a closed-loop control, the digital delay will be defined as the time elapsed from a measurement (voltage, current, etc.) gets sensed until an associated switch operation get taken. The digital delay is more critical to closed-loop system since it will affect the system transient response and even stability.

Figure 6-15 (a) shows if at the beginning of a switching period, the UC reading in all necessary sensing data for closed-loop computation, which is actually acquired at HMs in the previous switching period, and then generates switch commands and send to every HMs. These commands will decide the action of PEBBs at the beginning of next switching period. Thus, the digital delay in this case will be 2 switching periods. If data sensing, ADC conversion at HMs and UC reading sending data, computing and sending commands to HMs can be arranged in one switching period, then the digital delay can be reduced to one switching period, as shown in Figure 6-15. In the former case, UC could have longer time for computation; while in the latter case, the control loop can have higher bandwidth and better performance.



(a) 2Ts delay



(b) Ts delay

Figure 6-15. Different digital delay caused by communication.

6.2.3 Experimental Results

Two control applications were verified on the 3-phase PEBB-based converter system – open loop SPWM VSI and closed-loop VSI. The applications were simulated first in Saber, with the circuit diagram shown in Figure 6-16.

When the switching frequency is 20 kHz, considering the stringent timing requirements, the minimal footprint DARK option–mailbox and single thread static scheduling – is chosen. Figure 6-17 shows the testing waveforms of the open loop SPWM application.

For the current closed-loop VSI application, a proportional-integration (PI) integrator is used as the compensator. The control design is based on average small signal model in continuous time domain, and then discredited. Again for DARK, mailbox and single thread static scheduling is chosen as the real-time option. Two times switching periods is considered as the digital delay. Figure 6-18 shows the testing waveforms with step-up and step-down transient response respectively.

With the improved PESNet protocol, one switching period delay is achievable. The transient tests were repeated for this case. However, if the switching frequency is 20 kHz, the computation time will be too stringent for the $1*T_s$ digital delay case. For a fair comparison in order to show the impact on system performance of digital delay, all the rest tests are carried with $f_s = 10$ kHz. The shorter the digital delay, the shorter computation time, which means DARK has to run with lower footprint. On the other hand, the shorter the digital delay, the higher system bandwidth can be achieved and hence faster transient response. Figure 6-19 shows the test waveforms for the same VSI working at 10 kHz switching frequency. Figure 6-20 shows the transient experimental results under $2*T_s$ digital delay. The DARK option is single thread static scheduling and mailbox styled data channel. Figure 6-21 shows the transient experimental results under T_s digital delay. The DARK option is single thread dynamic scheduling and mailbox data channel.

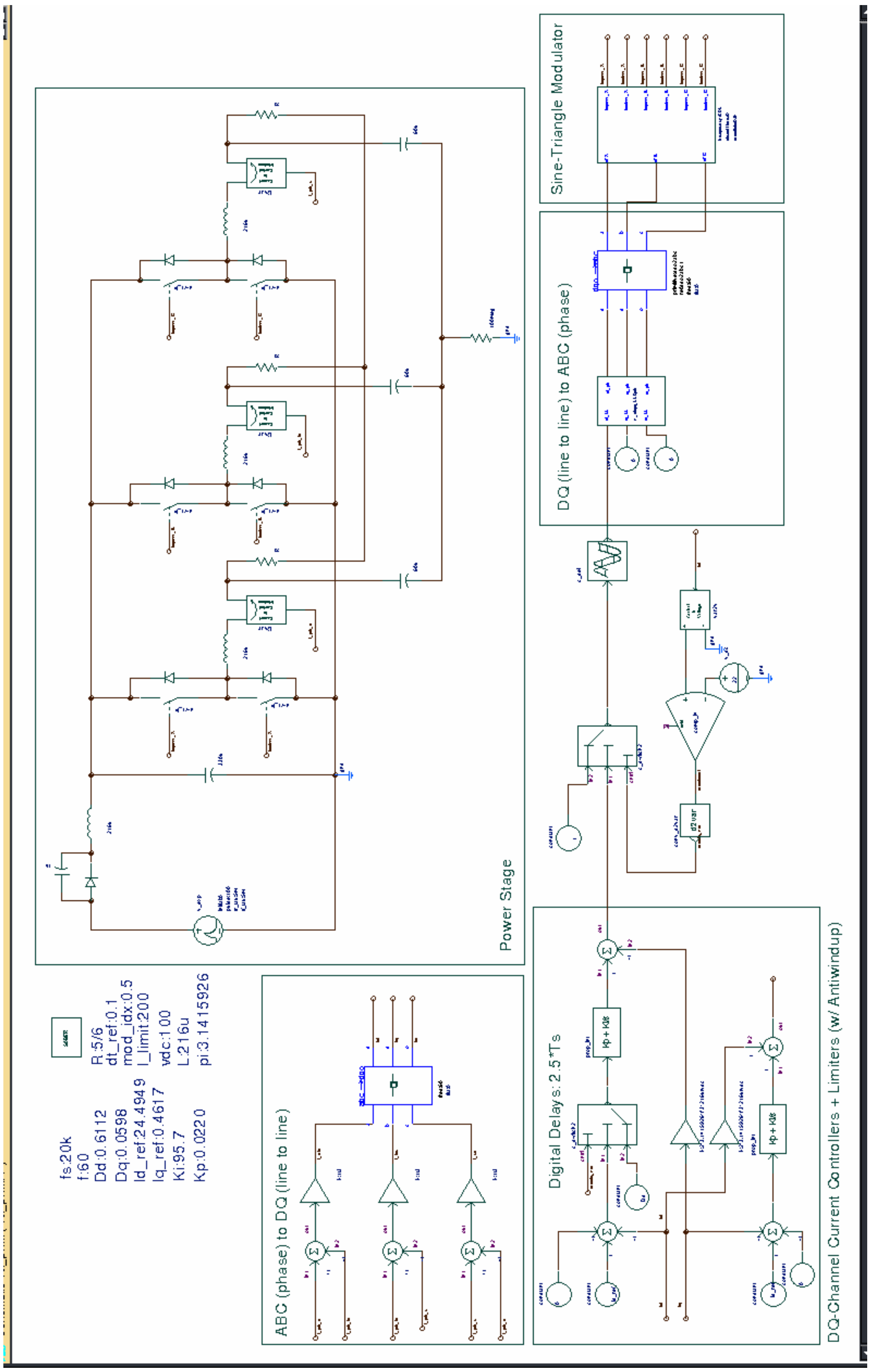


Figure 6-16. Saber simulation.

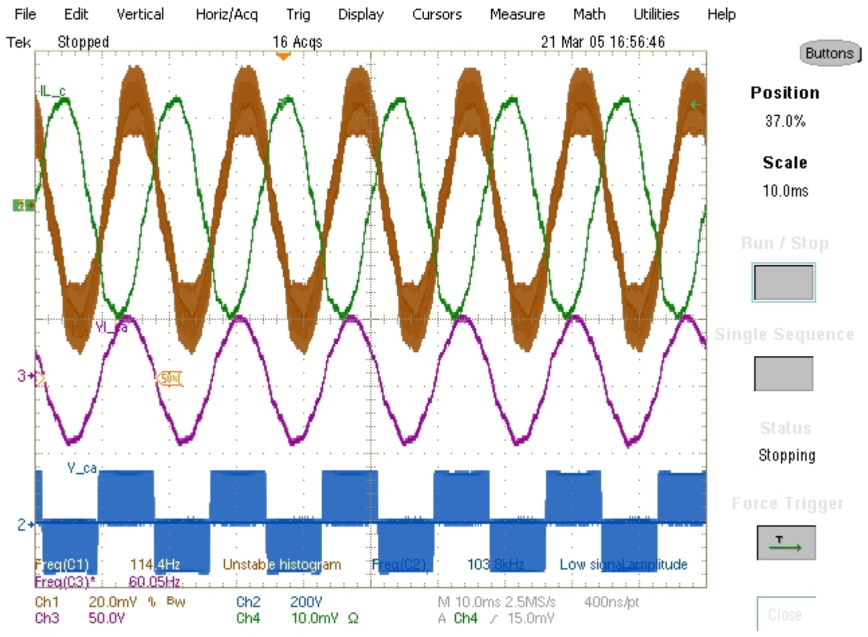
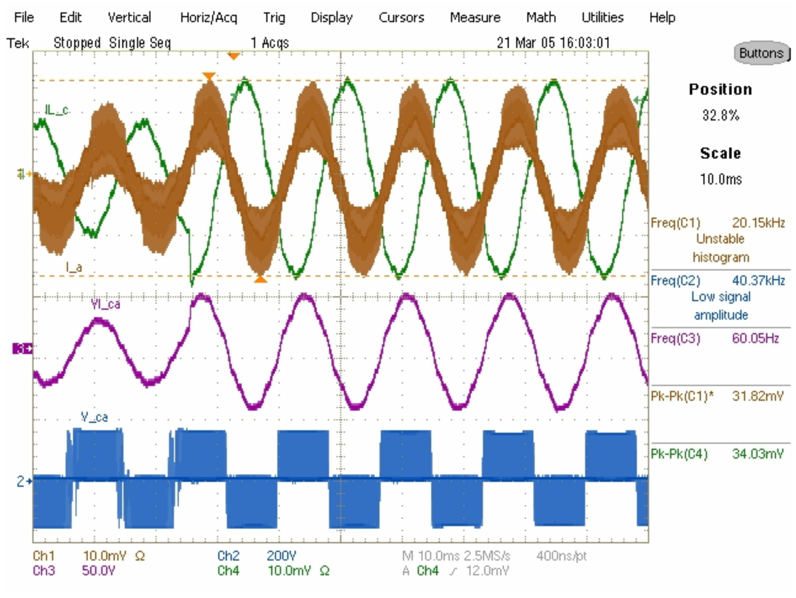
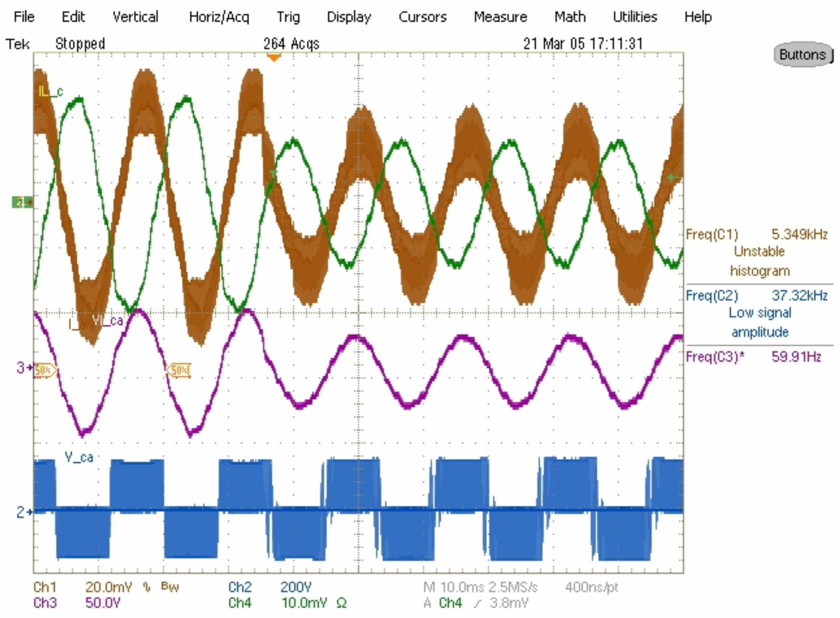


Figure 6-17. Open loop test waveforms.

Figure 6-17 shows inductor current, which is in brown, phase load current, which is in green, line-to-line voltage, which is in green and switch node voltage, which is in blue. Both in the voltage and current waveforms, distortions can be observed at waveform peaks and zero-crossings. Par of those distortions are due to the relatively long deadtime implementation at the HM. The IPM used does not have built-in deadtime function. And the necessary deadtime control is implemented in the FPGA of HM. The rise time and falling time if the IPM module is around 1 us, and the deadtime for each transition is designed to be 5 us for safety consideration. When the switching frequency is pushed to 20 kHz, 5 us will be 10% of the switching period, which could cause significant distortion when the duty cycle reaches it minimum or maximum values, or when the phase current is changing its direction. This kind of distortion occurs with highest frequency up to switching frequency, and will be hard to be reduced through traditional feedback control. The deadtime compensation at the HM can alleviate this issue, but will not get rid of it.



(a) Step-up



(b) Step-down

Figure 6-18. Closed-loop test waveforms.

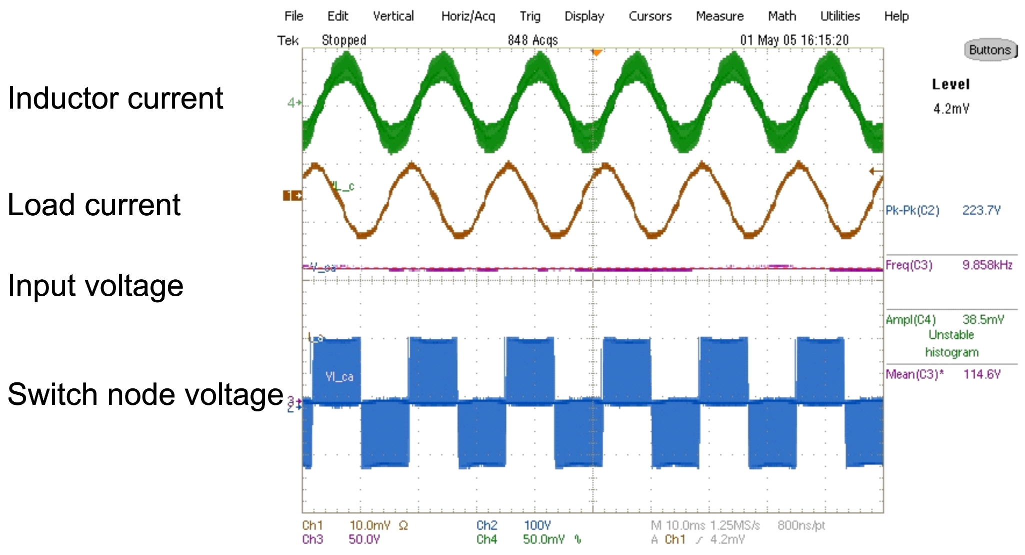
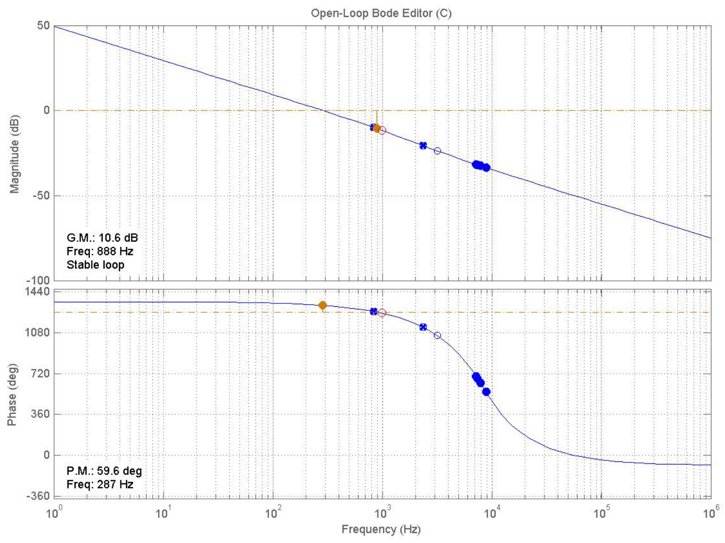
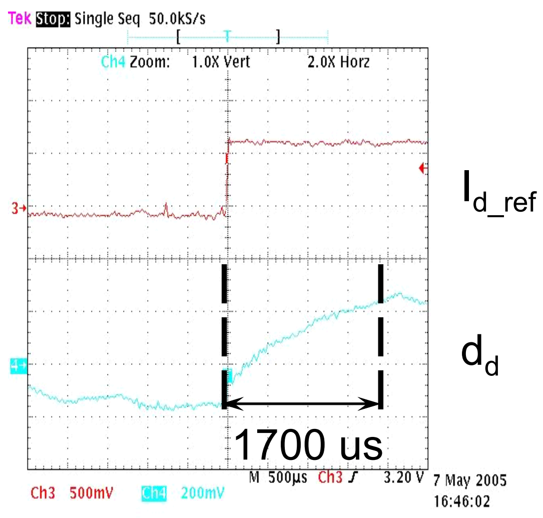

Inductor current
Load current
Input voltage
Switch node voltage
File Edit Vertical Horiz/Acq Trig Display Cursors Measure Math Utilities Help
Tek Stopped 848 Acqs 01 May 05 16:15:20
Buttons
Level 4.2mV
Pk-Pk(C2) 223.7V
Freq(C3) 9.858kHz
Ampl(C4) 38.5mV Unstable histogram
Mean(C3)* 114.6V
Ch1 10.0mV Ω Ch2 100V M 10.0ms 1.25MS/s 800ns/pt
Ch3 50.0V Ch4 50.0mV A Ch1 ⁄ 4.2mV

Figure 6-19 Open loop test waveforms for f_s = 10 kHz.

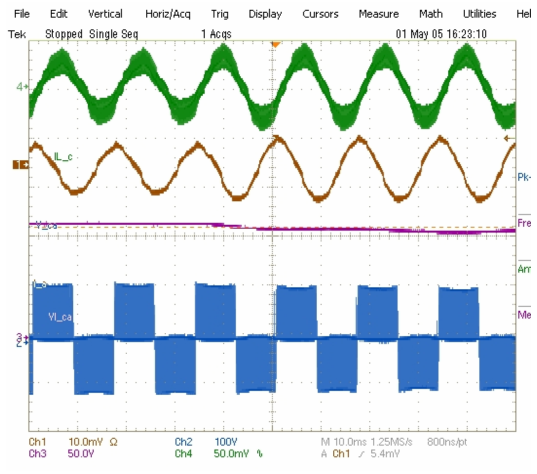
The control network latency affects system performance, especially the transient response. With longer delay, the lower the control bandwidth regarding to achieve the same gain margin (G.M.) and phase margin (P.M.), as shown in Figure 6-20.



(a) Loop gain design for t_{delay}=2*T_s, f_c = 287 Hz.

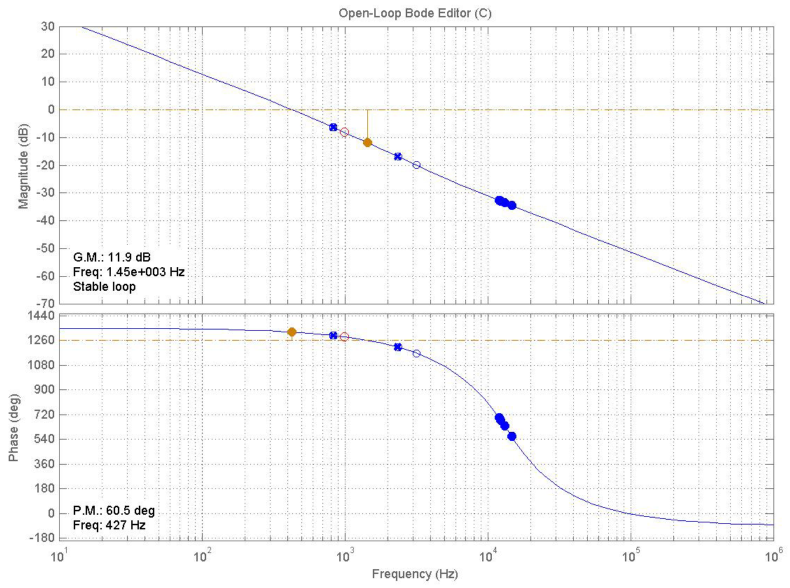


(b) Measurement of transient response.

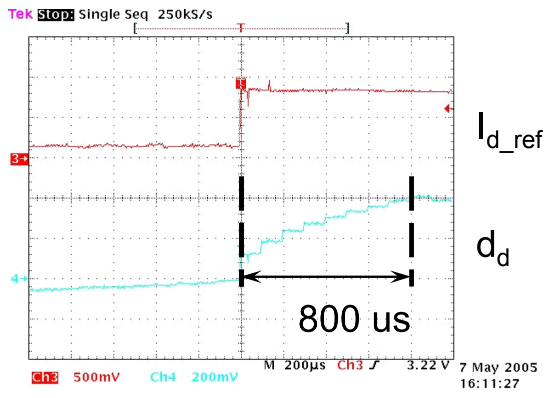


(c) Test waveforms during step up transient.

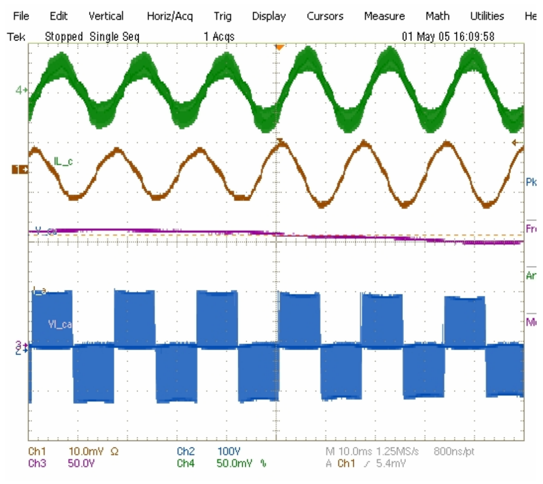
Figure 6-20. Control design and transient response for t_{delay} = 2*T_s.



(a) Loop gain design for t_{delay}=T_s f_c = 427 Hz.



(b) Measurement of transient response.



(c) Test waveforms during step up transient.

Figure 6-21. Control design and transient response for $t_{delay} = T_s$.

Chapter 7: FUTURE DIRECTION AND CONCLUSIONS

7.1 Towards Reconfigurable Control Software

If a fault or exception is detected by the system hardware, the simplest way to handle it is to shut down the whole system. This is a safe practice, but in some cases is an overreaction. For example, if only one current sensor is malfunctioning but the all power switches and gate drivers can still conduct normal operations, it is possible to modify the control to continue without current loop, instead of shutting down. However, if one of the phases fails in the multi-phase converter system, or an over current or over voltage situation is detected, it is probably safer to shut down the whole power stage. From the two examples presented above, it is desirable for the control to provide certain level of adaptability to failures or changes in the power hardware.

Shutting down the system is still a common practice in case of any hardware failure. One reason from the software point of view is there no easy way to implement online reconfiguration if the control software is not well structured.

In the dataflow architecture, the ECOs are executed independently and solely connected through data channels, and a control application is defined by a dataflow graph. These properties not only reduce the engineering effort for control software development and redesign, but also open the possibility from software point of view to reconfigure the control software on the fly to adapt to changes in the outside world without introducing massive complexity to the software development.

7.1.1 Reconfiguration Mechanism

The open control architecture proposed in Chapter 3 provides a feasible path for implementing reconfiguration without involving too much complexity to the control software composition. In the open control architecture, the relationship between control software and its controlled power hardware is encapsulated in the application layer. If the reconfiguration can also be performed in the application layer, this will leave all the lower level layers and upper level layers unaffected. The basic procedure of control reconfiguration is shown in Figure 7-1. The control tracks the status of every component in the power hardware. Once there is a status change, the control will apply certain rules to decide whether a software transition is necessary and what transition should take. The rest of the discussion assumes that the system has the structure shown in Figure 7-3. The design of control software reconfiguration includes:

1. Representation of power hardware status in control software;
2. Track of power hardware status;
3. Rules of control software reconfiguration according to power hardware changes; and
4. Software reconfiguration implementation.

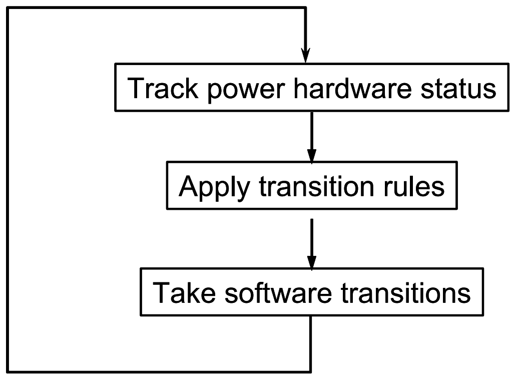


Figure 7-1. Procedure of software configuration.

7.1.1.1 Track Power Hardware Status

Since the reconfiguration is based on the power hardware changes, the control software must keep the up-to-date hardware status. Figure 7-2 shows the table in the application layer used for the tracking of power hardware status.

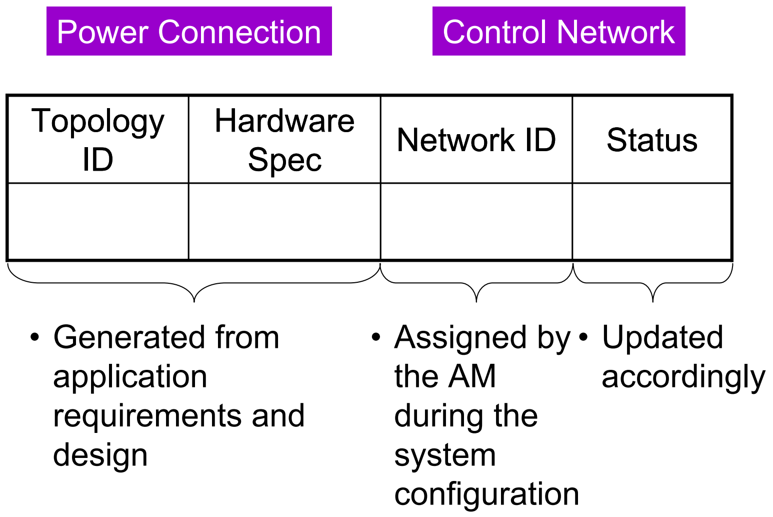

Power Connection
Control Network
Topology ID
Hardware Spec
Network ID
Status
• Generated from application requirements and design
• Assigned by the AM during the system configuration
• Updated accordingly

Figure 7-2. Current hardware status table

In the hardware status table, the "*Topology ID*" indicates the position and function of a hardware component in the whole power processing system. Typical hardware components in the multi-phase converter system can be phase leg, current sensor or voltage sensor. But only the hardware type cannot uniquely express the function of a hardware component in a converter system to implement certain application. For example, even a same voltage sensor can be used to measure DC link voltage in one application, and phase voltage in another application. "*DC voltage sensor*" and "Phase A voltage sensor" are two different topology ID.

"*Hardware Spec*" is used to describe the specification of every hardware component. For a voltage sensor, the typical specification may include the sensing range, conversion rate, offset, etc. With the specification, the control should be able to decide whether a hardware component can serve the role that the "*Topology ID*" indicates.

When an application is defined, the control application designer should have the list of all the hardware components that will serve for the application. During the configuration stage after the system starts up, if the control software also has chance to enquire specifications from every hardware component, it can then decide whether a hardware component is suitable in the application.

If the system is composed from distributed hardware nodes, the control intelligences may communicate with each other and every hardware component is assigned a "Network ID". It should be noticed that several hardware components may have the same "Network ID". For example, in the PEBB-based converter system presented in previous chapter, the hardware components in one PEBB module – phase leg, voltage sensor and current sensor – will share the same network ID in PESNet.

The last column in the current hardware status table stands for the status of each component. In the simplest implement of software reconfiguration, the hardware status is defined as a Boolean type variable, which indicates that a hardware component is under either the two states – normal or failed. Suppose that every hardware component is able to report its status to the control software periodically, the control software should be able to capture changes of a hardware component: either from normal condition to failure condition or vise versa.

7.1.1.2 Define System States and Software Transition

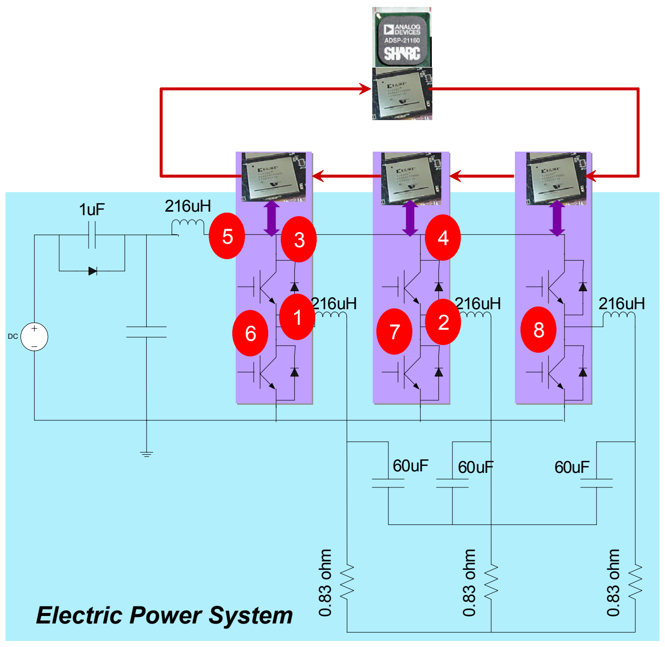
When every hardware component in a system is operational, the control software may perform with the full functionality as designed. When the system loses some hardware components, the control software may reconfigure itself to a degraded operational version. When the failed hardware component gets replaced or repaired, the control software may be able to switch back to it full featured version. Thus a system state is defined as a combination of a set of hardware components with normal conditions and a corresponding operational control software version. A software transition means the control software switch from one operational version to another as the system state changes. For conventional brute-force shut down approach to handle hardware exceptions and failure, the system only has two states: the normal state and the shutdown state. The control software only has two operational versions–the full featured version and the shut down commands. This research proposes to introduce more system stats and operational control software versions according to the nature of the power hardware and application.

Rules must be defined that when a hardware change is captured the system should stay in the current state or should change to another state. In this research work, all the software transition rules are predefined.

With the dataflow architecture proposed in this research work, all the software reconfiguration can be implemented in the control application layer without tackling individual ECOs, control host interface and network interface. The software transition can be achieved by activating/deactivating sets of ECOs.

7.1.2 Implementation of Control Software Reconfiguration under Dataflow Architecture

In this subsection, a 3-phase PEBB-based inverter system will be used as a design example to show the implementation of software reconfiguration. Figure 7-3 shows the system structure. The position of every hardware component is also marked. With the presence of DC link voltage sensor and phase voltage and current sensors, the full featured control for this converter can be a dual loop control: a DC link voltage feed forward control loop and a current feedback control loop. The dataflow control block diagram that incorporates the two control loops is shown in Figure 7-4. The DC link voltage is sensed, the DC link voltage error after regulation will be used as reference in the phase current feedback loop. Table 7-1 shows a possible current hardware status after system start up, which supposes that after the system startup, every hardware component is operational. In the communications network, there are one UC and three HMs, and assigned network address 0x01, 0x02, 0x03, 0x04, respectively. The dataflow styled control software is running on UC. It also supposes that with the support of the communications protocol PESNet, the status of every hardware component is been able to report to UC periodically.



Hardware components:
1. Phase A current sensor
2. Phase B current sensor
3. Phase A voltage sensor
4. Phase B voltage sensor
5. DC bus voltage sensor
6. Phase A
7. Phase B
8. Phase C

Figure 7-3. A PEBB-based 3-phase inverter with hardware component list and position.

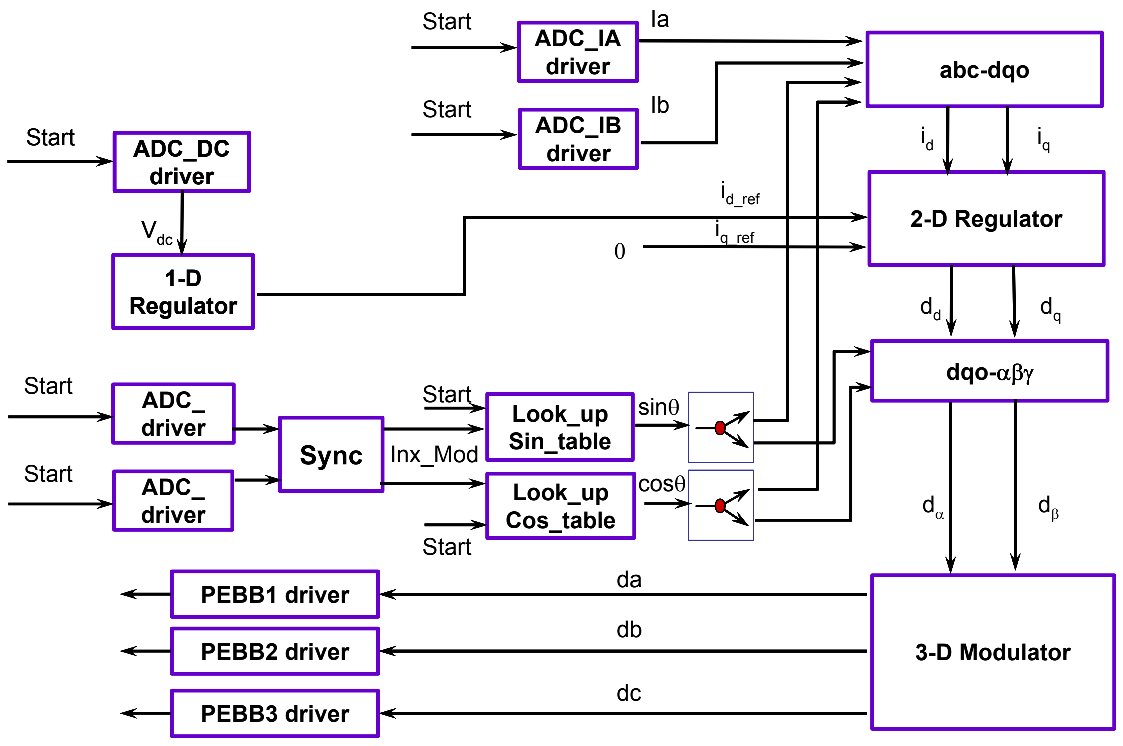


Figure 7-4. Dataflow graph of a 3-phase inverter control with DC link voltage feed forward and phase current feedback control loops.

Table 7-1. Current Hardware Status.

Topology ID	***Spec***	***Network ID***	***Status***
0x01	Current sensor; 12-bit	0x02	Normal
0x02	Current sensor; 12-bit	0x03	Normal
0x03	Voltage sensor; 12-bit	0x02	Normal
0x04	Voltage sensor; 12-bit	0x03	Normal
0x05	Voltage sensor; 12-bit	0x04	Normal
0x06	Phase leg; half-bridge	0x02	Normal
0x07	Phase leg; half-bridge	0x03	Normal
0x08	Phase leg; half-bridge	0x04	Normal

Table 7-2. Four operational system states for the 3-phase inverter application.

System State	***Hardware Status***	***Applicable Control Algorithm***
S0	All hardware components working in normal condition	DC link voltage feed forward + phase current feedback control
S1	DC link voltage sensor fails; other hardware components are normal	Phase current feedback control
S2	Any current sensors fails; other hardware components are normal	Open-loop control
S3	Any phase leg fails; other hardware components are normal	Shut down

Table 7-2 lists the four operational system states according to the presence of hardware components. Under the dataflow architecture, every state has an applicable control algorithm and therefore has a corresponding dataflow graph. When system loses some hardware components, the system can change from one state to another; also when the failed hardware component(s) is repaired or replaced, the system stats may also change. The possible transitions between the four states S0-S4 caused by hardware status changes is shown in Figure 7-5. Every arrow between two states is a transition. For example:

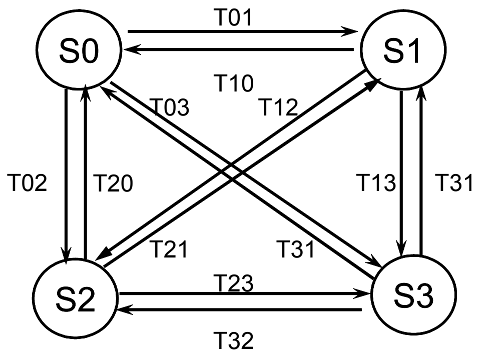


Figure 7-5. State transitions.

T01: lose DC link voltage senor;

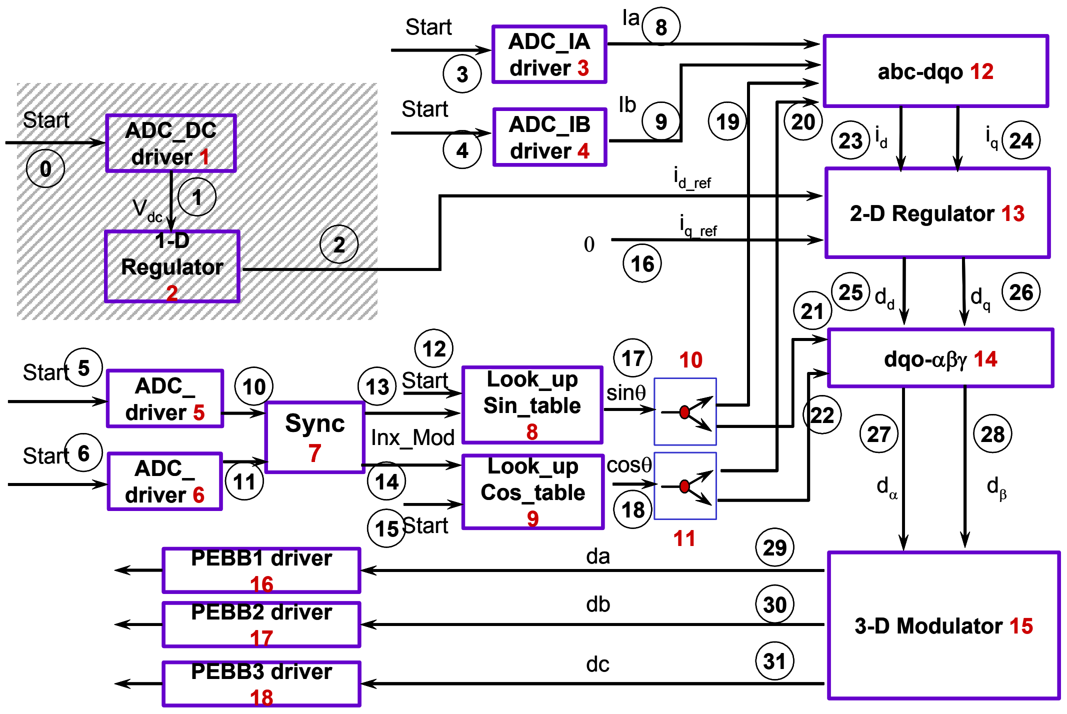
T10: DC link recovers;

T02: lose any current sensor;

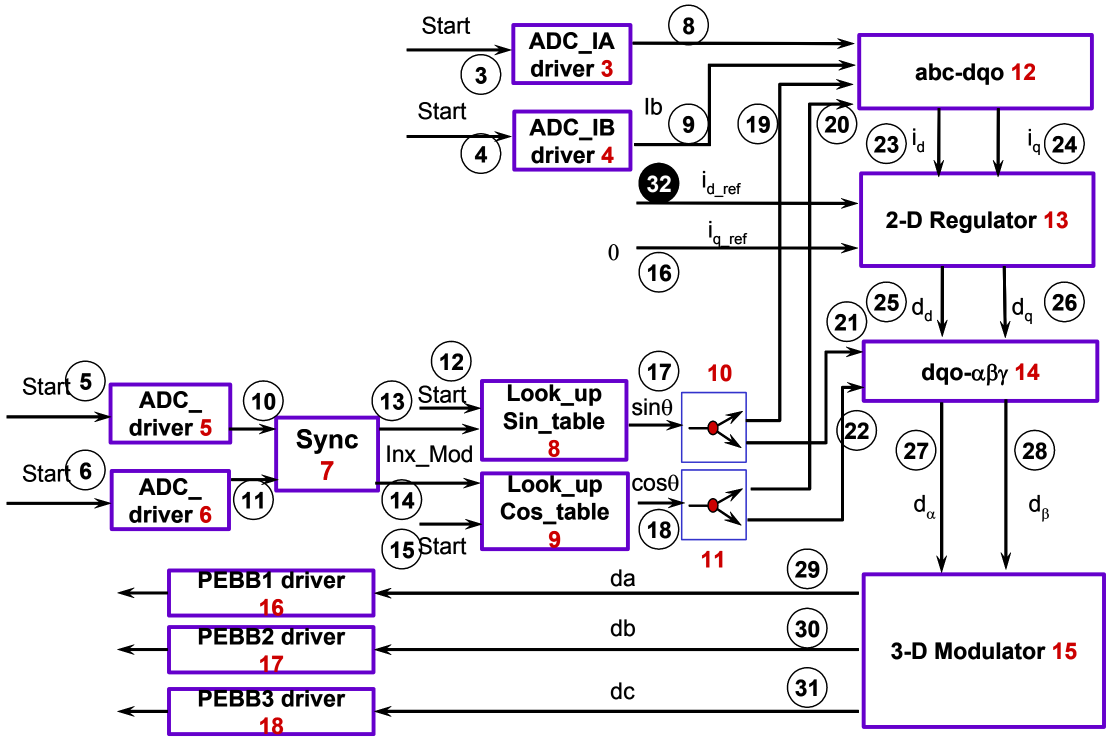
T20: the failed current sensor recovers;

….

Every system state has a corresponding dataflow graph to represent the applicable control algorithm. If all the hardware components are working in normal condition, the system is state S0 and the applicable control can be a DC link feed forward loop and a phase current feedback loop, as shown in Figure 7-6 (a). The feed forward loop is supposed to help on system transient response to line voltage variations. In the real code implementation of a data flow graph, every ECO process possesses a unique ECO ID, and the same for data channels.



(a) Working dataflow graph (DFG0) when all hardware components are operational.



(b) Working dataflow graph (DFG1) after DC link voltage sensor is not operational.

Figure 7-6. Dataflow graph transition caused by the loss of the DC link voltage sensor.

If the DC link voltage sensor gets failed, the feed forward sensing data of the DC link voltage is invalid and the system is in state S1. Thus, the feed forward loop should be excluded from the control algorithm, while the current feedback control loop is still conductible if the current reference is set to be a fix value instead of taking the result from the feed forward voltage control loop in previous case. The corresponding dataflow graph is shown in Figure 7-6 (b). However, the performance of control is degraded. This transition T01 (S0→S1) is caused by the loss of the DC link voltage sensor. The reconfiguration of control software can be implemented by switching the dataflow graph under execution from DFG0 to DFG1, which means the following:

- In the ECO processes set, ECO 1 and ECO 2 should be deactivated;
- In the data channel set, data channels 0, 1, 2 should be deactivated, and data channel 32 should be activated.

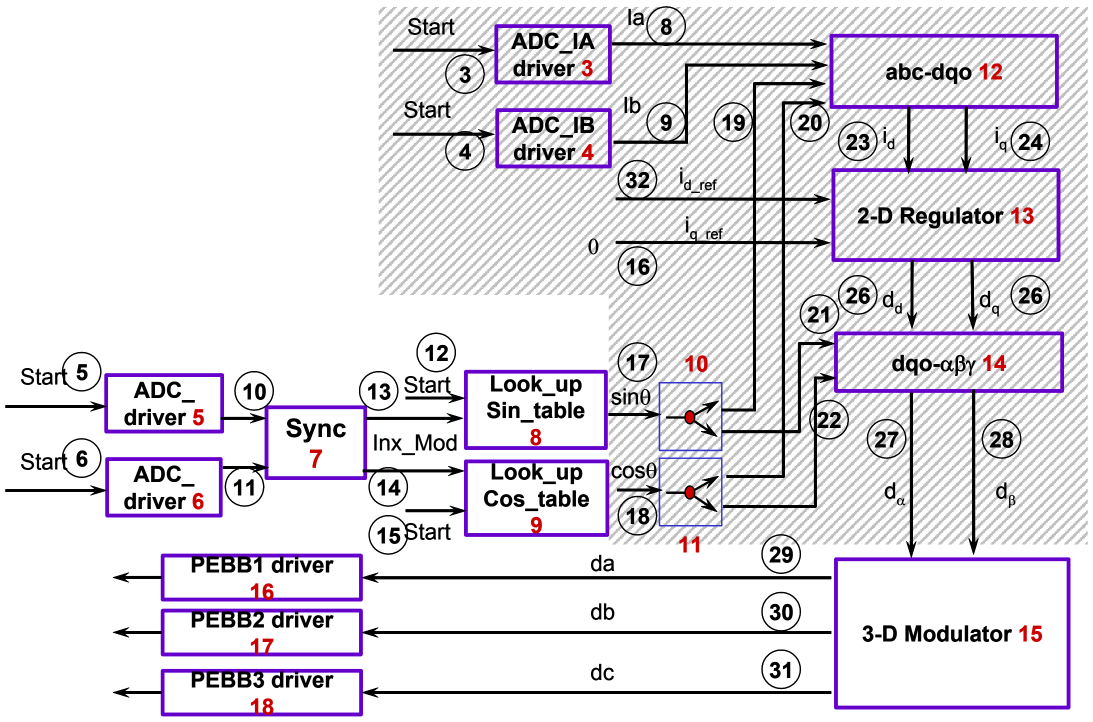
The changing part in the dataflow graph is shaded in Figure 7-6 (a).

In the case of any current sensor failure, the system will be in state S2 and the control will have to be degraded further to an open loop version, as shown in Figure 7-7 (b). The reconfiguration of control software can be implemented by switching the dataflow graph under execution from DFG1 to DFG2, which means the following:

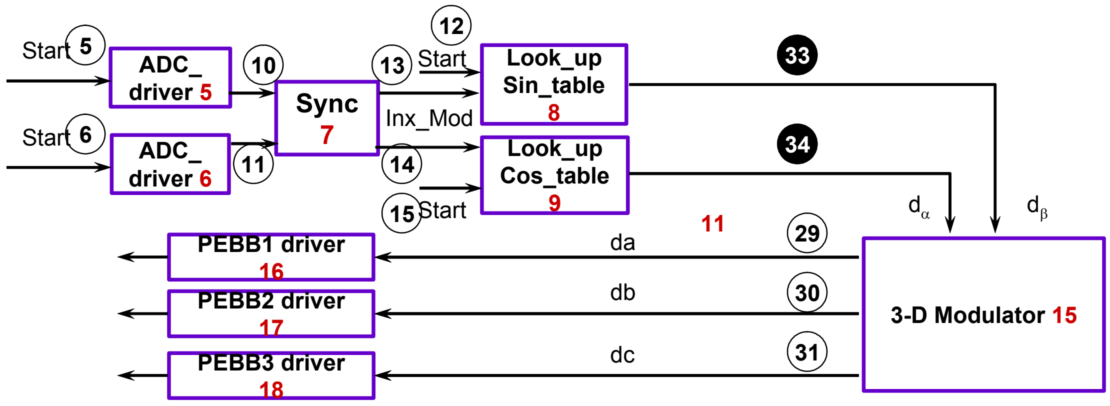
- In the ECO processes set, deactivated ECOs 3, 4, 10, 11, 12, 13, 14;
- In the data channel set, deactivated 3, 4, 8, 9, 16-28, 32, and activate 33 and 34.

In the case of any phase leg failure, for example reported over current or over voltage in any power module, the system will be in state S3 and the control should shut down all switches in order to prevent further damage to the system.

When the system state transition occurs, the control can be reconfigured through the transition of dataflow graphs. For example, when transition T01 happens, the system state changes from S0 to S1. For the control, DFG1 should substitute DFG0 to be the working dataflow graph. The transition of dataflow graphs can be implemented through deactivating the ECOs and data channels, as shown in Figure 7-7 (a) and (b).



(a) Working dataflow graph (DFG1) when the system loses the DC link voltage sensor.



(b) Working dataflow graph (DFG2) after the system loses any one of the current sensors.

Figure 7-7. Dataflow graph transition caused by the loss of any one of current sensors.

Since every state is related to hardware status, the transition rules between states can be defined according to the presence and absence of hardware components. Every state is associated with two masks – a hardware presence mask and a hard absence mask. In this design example 8-bit binary array will be used as mask. In the presence mask of a state Si, '1' at jth bit means a hardware component with topology ID j has to be in normal condition for state Si. '0' means optional. In the absence mask of a state Si, '1' at kth bit means a hardware component with topology ID k is required to be in failed condition. For S0, it requires that all hardware components are normal. So the presence mask for S0 will be:

Mask(0, 0) = b11111111= 0xFF.

The absence mask for S0 will be:

Mask(0,1) = b00000000 = 0x00.

Similarly, the masks for other states can be derived, and all the masks can be expressed as a two-dimensional binary array:

```
Mask[4][2] = {
    {0xFF, 0x00};
    {0xF7, 0x08};
    {0x07, 0xF8};
    {0x00, 0x07}
}
```

An 8-bit binary array (Current_Hardware_Status) can be used to represent status of the 8 hardware components, in which '1' means normal, and '0' mean failed. Every time when the Current_Hardware_Status gets changes, it will be compared to the pair of masks of each state as follows:

- ***Check 1:*** (mask1 & new_sys_state) = = mask
- ***Check 2:*** (mask2 & !new_sys_state) ! = 0

If both checks are satisfied for a state, the state will be the next state of the system. According to the current state and next state, whether a transition is necessary and the reconfiguration actions can be uniquely decided. Since for different state there is a unique dataflow graph associated with, the reconfiguration action for a system state transition can be interpreted into activation of a certain set of ECOs and data channels and deactivation of another set of ECOs and data channel. A reconfiguration matrix is such a table that records reconfiguration actions for every transition defined for the system. The declaration of the reconfiguration matrix is:

Table 7-3. Data structure declaration of reconfiguration action

```
typedef struct
{
        Transition_Priority priority;
        int *deactivate_eco;
        int *deactivate_dc;
        int *activate_eco;
        int *activate_dc;
}Reconfig_Data;
```

The transition priority means when more than one transition is possible, which transition will have the highest priority. For example, the reconfiguration action for transition S0→S1 will be defined as:

Table 7-4. Reconfiguration action for transition S0 → S1.

```
Reconfig_action[0][1] = {
        0; // transition priority
        {1, 2}; //deactivated ECOs
       {0, 1, 2}; //deactivated DCs
        {}; //activated ECOs
        {32}; //activated DCs
};
```

Thus, a reconfiguration matrix can be defined as:

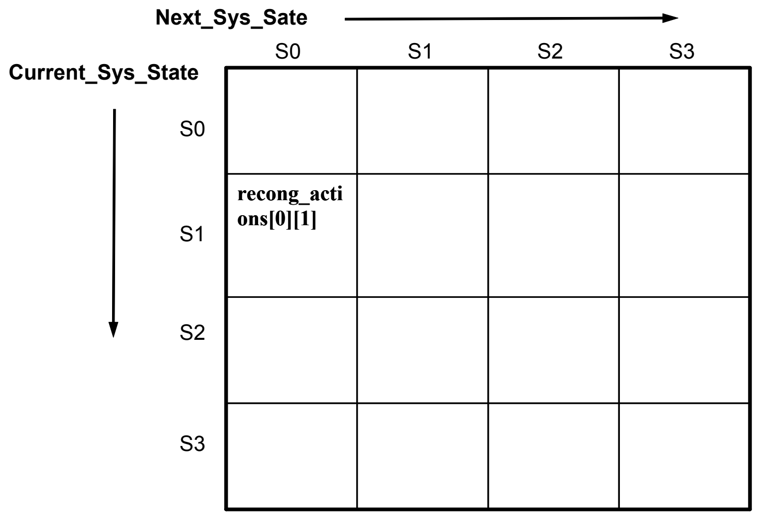


Figure 7-8. Reconfiguration matrix.

7.1.3 Further Issues Related to Online Control Reconfiguration

So far the feasibility of control reconfiguration was only been considered from the software implementation point of view. There are still several issues that need to be clarified from control point of view.

The first question is fault and exception detection. For some kinds of hardware failures, such as over current or over voltage for a power module, the detection is easy and normally a built-in function of commercialized products. But definitions for some other kinds of hardware failures are difficult or ambiguous. For example, if a reading from a sensor is abnormal, it is hard to tell whether it is due to the malfunction of the sensor or of the sensed object. The failures and exceptions in a communications network are even more complicated.

The second issue the latency of control reconfiguration. Some time will be required for the control software to reconfigure. We can define the reconfiguration latency as the time elapsed from a failure occurs until the control software responds. The question then will be how fast the control could respond to a hardware failure and what is the tolerance of this reconfiguration latency.

Another critical issue is how control reconfiguration will affect the overall system stability and whether possible transients due to control software reconfiguration are tolerable to the hardware system. In other words, for all the software components related to post-reconfiguration control and also incorporate states, how to set the values of those states to prevent system instability and help transient.

The last issue is how one can simulate dataflow control software and reconfiguration cases to analyze the performance of control software and effects of software reconfiguration.

7.2 Conclusions

Due to close coupling between control software and hardware, and to a lack of standardization, the legacy proprietary approach to real-time embedded control software design in power conversion systems has a number of problems. After analyzing real-time embedded control from various aspects–functionality, real time, and interactions with the real world–this research work has proposed an open control architecture to localize and encapsulate control software design dependencies from other areas in power conversion systems in order to reduce the overall design time and cost of software development. This solution provides a better mechanism for managing complexity within the problem domain. Based on the exploration of mainstream software styles, the dataflow style is chosen for open control architecture. To assess the feasibility of dataflow-based control software in power conversion systems, four design applications are implemented in this style; these range from a relatively simple open-loop, SPWM three-phase inverter to a fairly complex four-leg inverter. The levels of performance for dataflow software are compared to both handwritten main-program-and subroutine code and generated code from SimuLink + Real-time Workshop.

The advantages of the dataflow-based open control architecture can be summarized as:

- It leads naturally to a component-based design approach for control software;
- It provides a framework to support modularity, reusability and reconfigurability for software designers;
- It reduces overall development cost and time;

- It decreases the complexity of development and testing;
- It minimizes redesign effort;
- It allows designers to focus easily on key design areas;
- It facilitates system expansion and upgrades.

The main disadvantage of the dataflow architecture is the performance overhead it incurs. Initially, this overhead might be considered to be far too great for dataflow to be practical. The preliminary study reported here indicates otherwise, however. In addition, the use of hard real-time scheduling strategies to pre-compute ECO execution sequences is a fruitful avenue for exploration, offering the potential for dramatic increases in performance.

REFERENCES

[1] Lee, K., Yao, K., Zhang, X., Qiu, Y. and Lee, F.C., "A novel control method for multiphase voltage regulators," Proc. of Eighteenth IEEE Applied Power Electronics Conference and Exposition, APEC 2003, Vol. 2, pp. 738 - 743, Feb.2003.

[2] Yao, K., Lee, K., Xu, M. and Lee, F.C., "Optimal design of the active droop control method for the transient response," Proc. of IEEE Eighteenth Applied Power Electronics Conference and Exposition, 2003. APEC '03 Annual, Vol. 2, 718 - 723, Feb. 2003.

[3] Zhou, X., Peng X. and Lee, F.C., "A high power density, high efficiency and fast transient voltage regulator module with a novel current sensing and current sharing technique," Proc. of IEEE Fourteenth Applied Power Electronics Conference and Exposition, 1999. APEC '99. Vol. 1, pp. 289 - 294, Mar. 1999.

[4] Rossetti N., "Valley design techniques outperform peak current-mode approach for CPU supplies," in PCIM, July 2001.

[5] Miftakhutdinov, R., "Analysis of synchronous buck converter with hysteretic controller at high slew-rate load current transients," in Proc. of HPFC, pp. 55-69, 1999.

[6] W. Gu, W. Qiu, W. Wu and I. Batarseh, "A multiphase DC/DC converter with hysteretic voltage control and current sharing," in Proc. IEEE APEC, 2002, pp. 670-674.

[7] W. Huang, "A new control for multi-phase buck converter with fast transient response," in Proc. IEEE APEC, 2001, pp. 273-279.

[8] du Toit, J.A., Enslin, J.H.R. and Spee, R., "Experimental evaluation of digital control options for high power electronics," Industrial Electronics, Control, and Instrumentation, 1995., Proceedings of the 1995 IEEE IECON 21st International Conference on, Vol. 1, Nov. 1995, pp.674-679.

[9] Norum, L., Sulkowski, W. and Aga, L.A., "Compact realisation of PWM-VSI current controller for PMSM drive application using low cost standard microcontroller," *Power Electronics Specialists Conference, 1992. PESC '92 Record., 23rd Annual IEEE*, Vol.1, Jun. 1992, pp. 680-685.

[10] Gyeong-Hae Han and Bum-Suk Ko, "All-digital logic control PWM/PFC for inverter system," *Electronics, Circuits and Systems, 2000. ICECS 2000. The 7th IEEE International Conference on*, Vol. 2, Dec. 2000, pp. 789 – 792.

[11] De Gussemd, K., Van de Sype, D.M., Van den Bossch, A.P. and Melkebeek, J.A., "Digital control of boost PFC converters operating in both continuous and discontinuous conduction mode," *Power Electronics Specialists Conference, 2004. PESC 04. 2004 IEEE 35th Annual*, Vol. 3 , Jun. 2004, pp. 2346-2352.

[12] Zhang, L., Watthanasarn, C. and Shepherd, W., "Analysis and comparison of control techniques for AC-AC matrix converters," *Electric Power Applications, IEE Proceedings*, Vol. 145, Issue: 4, Jul. 1998, pp. 284-294.

[13] Angkititrakul, S. and Erickson, R.W., “Control and implementation of a new modular matrix converter,” *Applied Power Electronics Conference and Exposition, 2004. APEC '04. Nineteenth Annual IEEE*, Vol. 2, 2004, pp. 813-819.

[14] du Toit, J.A., Bester, D.D. and Enslin, J.H.R., “A DSP based controller for back to back power electronic converters with FPGA integration,” *Applied Power Electronics Conference and Exposition, 1997. APEC '97 Conference Proceedings 1997, Twelfth Annual,* Vol. 2, Feb. 1997, pp. 699-705.

[15] du Toit, J.A. and Enslin, J.H.R., “A DSP based controller for high dynamic bandwidth power electronic applications,” *AFRICON, 1996., IEEE AFRICON 4th,* Vol. 2, Sept. 1996, pp. 665-670.

[16] Malesani, L., Rossetto, L. and Zuccato, A., “Digital adaptive hysteresis current control with clocked commutations and wide operating range,” *Industry Applications, IEEE Transactions on*, Vol. 32, Issue 2, Mar.-Apr. 1996, pp. 316-325.

[17] F. Vahid and T. Givargis, “Embedded System Design: A Unified Hardware/Software Introduction,” *John Willey & Sons, Inc.,* 2002.

[18] Jae-Ho Choi and Byoung-Jin Kim, “Improved digital control scheme of three phase UPS inverter using double control strategy,” *Applied Power Electronics Conference and Exposition, 1997. APEC '97 Conference Proceedings 1997, Twelfth Annual*, Vol. 2, Feb. 1997, pp. 820-824.

[19] Bedford, B. and Hoft, R., “Principles of Inverting Circuits,” NY: John Wiley & Sons, Inc., New York 1964.

[20] L. Gyugyi and B. Pelly. “Static Power Frequency Changers: Theory, Performance and Application,” New York, NY: Wiley-Interscience, 1976.

[21] R. Lye, “Power Converter Handbook-Theory, Design, Applications,” Canadian General Electric Company Limited, 1976, Canada.

[22] T. Ericsen, A. Tucker, D. Hamilton, G. Campisi, C. Whitcomb, J. Borraccini and W. Jacobesen, “Chapter 5.2, Standardized Switch System Modules (Power Electronic Building Blocks),” Proceedings Of PCIM’97, Sep. 9-13, 1997, pp. PEO3.5,1-21.

[23] T. Ericsen and A. Tucker, “Power Electronics Building Blocks and Potential Power Modulator Applications,” Proceedings of PMS’98, 22-25 June 1998, pp. 12-15.

[24] T. Ericsen, “Power Electronics Building Blocks – a Systematic Approach to Power Electronics,” Proceedings of IEEE PES’98, vol. 2, 2000, pp. 1216-1218.

[25] K. Xing, et al, “Power electronics building block and system integration,” Project annual report submitted to Office of Naval Research, Center for Power Electronics Systems, Blacksburg, 1999.

[26] I. Milosavljevic, D. Borojevi and I. Celanovic, “Modularized Communication and Control Structure for Power Converters,” 8th European Conference on Power Electronics and Applications, EPE, September 1999.

[27] Celanovic, I., Milosavljevic, I., Boroyevich, D., Cooley, R. and Guo, J., "A new distributed digital controller for the next generation of power electronics building blocks," *Applied Power Electronics Conference and Exposition, 2000. APEC 2000. Fifteenth Annual IEEE*, Vol. 2, Feb. 2000, pp. 889-894.

[28] Celanovic, I., Celanovic, N., Milosavljevic, I., Boroyevich, D. and Cooley, R., "A new control architecture for future distributed power electronics systems," *Power Electronics Specialists Conference, 2000. PESC 00. 2000 IEEE 31st Annual*, Vol. 1, Jun. 2000, pp. 113 – 118.

[29] V. Vlatkovic and D. Borojevic, "Digital-signal-processor-based Control of Three-phase Space Vector Modulated Converters," IEEE Transactions on Industrial Electronics, vol. 41, no. 3, June 1994, pp.326 – 332.

[30] Bester, D.D., du Toit, J.A. and Enslin, J.H.R., "High performance DSP/FPGA controller for implementation of computationally intensive algorithms," *Industrial Electronics, 1998. Proceedings. ISIE '98. IEEE International Symposium on*, Vol. 1, July 1998, pp. 240-244.

[31] du Toit, J.A., le Roux, A.D. and Enslin, J.H.R., "An integrated controller module for distributed control of power electronics," *Applied Power Electronics Conference and Exposition, 1998. APEC '98. Conference Proceedings 1999, Thirteenth Annual*, Vol. 2, Feb. 1998, pp. 874-880.

[32] Joos, G. and Ziogas, P.D., "On maximizing gain and minimizing switching frequency of delta modulated inverters," *Industrial Electronics, IEEE Transactions on*, Vol. 40, Issue 4, Aug. 1993, pp. 436-444.

[33] OPCoDe project introduction, http://abb.com .

[34] LabView introduction, http://www.ni.com/aap/.

[35] B.K. Bose, "Power Electronics and Variable Frequency Drives," IEEE Press, 1996.

[36] B.K. Bose, "Power Electronics and Electrical AC Drives," Prentice Hall, Englewood Cliffs, NJ, 1986.

[37] A. Schonung and H. Stemmler, "Static Frequency Changers with Subharmonic Control in Conjunctionwith Reversible Variable Speed AC Drives," Brown Boweri Rev. 51, 1964, pp. 555-577.

[38] Venkataramanan, G., Divan, D.M., and Jahns, T.M., "Discrete pulse modulation strategies for high-frequency inverter systems," *Power Electronics, IEEE Transactions on*, Vol. 8, Issue 3, July 1993, pp. 279-287.

[39] Holtz, J., "Pulsewidth modulation for electronic power conversion," I, Vol. 82, Issue 8, Aug. 1994, pp. 1194-1214.

[40] Trzynadlowski, A.M., "An overview of modern PWM techniques for three-phase, voltage-controlled, voltage-source inverters," *Industrial Electronics, 1996. ISIE '96., Proceedings of the IEEE International Symposium on* , Vol. 1 , Jun. 1996, pp. 25-39.

[41] du Toit, J.A. and Beukes, H.J., “A distributed control strategy for multi-cell converters,” *Applied Power Electronics Conference and Exposition, 2001. APEC 2001. Sixteenth Annual IEEE,* Vol. 1, Mar. 2001, pp. 88-93.

[42] Habetler, T.G. and Divan, D.M., “Control strategies for direct torque control using discrete pulse modulation,” *Industry Applications, IEEE Transactions on*, Vol. 27, Issue 5, Sept.-Oct. 1991, pp. 893-901.

[43] Malesani, L. and Tomasin, P., “PWM current control techniques of voltage source converters-a survey,” *Industrial Electronics, Control, and Instrumentation, 1993. Proceedings of the IECON '93, International Conference on*, Vol. 2, Nov. 1993, pp. 670-675.

[44] Kazmierkowski, M.P. and Malesani, L.Current, “Control techniques for three-phase voltage-source PWM converters: a survey,” *Industrial Electronics, IEEE Transactions on*, Vol. 45, Issue 5, Oct. 1998, pp. 691-703.

[45] Trzynadlowski, A.M., “An overview of modern PWM techniques for three-phase, voltage-controlled, voltage-source inverters,” *Industrial Electronics, 1996. ISIE '96, Proceedings of the IEEE International Symposium on*, Vol. 1, Jun. 1996, pp. 25-39.

[46] van der Broeck, H.W., Skudelny, H.-C. and Stanke, G.V., “Analysis and realization of a pulsewidth modulator based on voltage space vectors,” *Industry Applications, IEEE Transactions on*, Vol. 24, Issue 1 , Jan.-Feb. 1988, pp.142-150.

[47] Kazmierkowski, M.P. and Dzieniakowski, M.A., “Review of current regulation methods for VS-PWM inverters,” *Industrial Electronics, 1993. Conference Proceedings, ISIE'93 - Budapest, IEEE International Symposium*, Jun. 1993, pp. 448-456.

[48] V. Blasko, “Analysis of a Hybrid PWM Based on Modified Space-Vector and Triangle-Comparison Methods,” IEEE Transactions Industrial Applications 33, 1997, pp.756-764.

[49] S. Ogasawara, H. Akagi and A. Nabae, “A Novel PWM Scheme of Voltage Source Inverter Based on Space Vector Theory,” In Conf. Rec. European Power Electronics Conf., 1989, pp.1197-1202.

[50] A. Hava, R. J. Kerkman and T.A. Lipo, “A High Performance Generalized Discontinous PWM Algorithm,” Proc. IEEE APEC, Atlanta,1997, pp. 886-894.

[51] J.A. Houldsworth and D.A. Grant, “The Use of Harmonic Distortion to Increase the Output Voltage of a Three-Phase PWM Inverter,” IEEE Transactions Industrial Applications., Sept./Oct. 1984, pp. 1224-1228.

[52] H. W. van der Broeck, H.Ch. Skudelny and G. Stanke, “Analysis and Realization of a Pulse Width Modulator Based on Voltage Space Vectors,” IEEE Transactions on Industrial Applications,24, 1998, pp. 142-150.

[53] G. Pfaff, A. Weschta and A. Wick, “Design and Experimental Results of a Brushless AC Servo Drive,” IEEE Transactions on Industrial Applications, IA-22, 1984, pp. 814-821.

[54] V. Blasko, “Analysis of a Hybrid PWM Based on Modified Space-Vector and Triangle-Comparison Methods,” IEEE Transactions Industrial Applications 33, 1997, pp. 756-764.

[55] S. R. Bowes and Y. S. Lai, “Relationship Between Space-Vector Modulation and Regular-Sampled PWM,” IEEE Transactions Industrial Electron. 44, 1997, pp. 670-679.

[56] F. Jenni and D. Wuest, “The Optimization Parameters of Space Vector Modulation,” Prceedings of EPE Conf., 1993, pp. 376-381.

[57] D. C. Lee and G. M. Lee, “A Novel Overmodulation Technique for Space-Vector PWM Inverters,” IEEE Transactions on Power Electron. 13, 1998, pp. 1141-1151.

[58] J. W. Kolar, H. Ertl and F.C. Zach, “Influence of the Modulation Method on the Conduction and Switching Losses of a PWM Converter System,” IEEE Transactions on Industrial Applications, 27, 1991, pp. 1063-1075.

[59] M. Malinowski, “Adaptive Modulator for Three-phase PWM Rectifier/Inverter,” Proceedings of EPE-PEMC Conference, Kosice, 2000, pp.135-141.

[60] H. van der Broeck, “Analysis of the Harmonics in Voltage Fed Inverter Drives Caused by PWM Schemes with Discontinuous Switching Operation,” Proceedings of EPE Conference, 1991, pp.261-266.

[61] D. M. Brod and D.W. Novotny, “Current Control of VSI-PWM Inverters,” IEEE Transactions on Industrial Applications, IA-21, 1985, pp. 562-570.

[62] J. Holtz, “Pulse Width Modulation for Electronic Power Conversion,” Proceedings of IEEE 82, 1994, pp.1194-1214.

[63] M.P. Kazmierowski and L. Malesani, “Spacial section on PWM current regulation,” IEEE Transactions Industrial Electron. 45, pp.689-802, 1998.

[64] M.P. Kazmierkowski and H.Tunia, “Automatic Control of Converter-Fed Drives,” Elsevier, Amsterdam, 1994.

[65] L. Malesani and P.Tomasin, “PWM Current Control Techniques of Voltage Source Converters—a Survey,” IEEE IECON’93, 1993, pp.670-675.

[66] A. M. Trzynadlowski, “Introduction to Modern Power Electronics,” John Wiley, New York, 1998.

[67] C. T. Rim, N. S. Choi, G. C. Cho and G. H. Cho, “A Complete DC and AC Analysis of Three-phase Controlled-current PWM Rectifier Using Circuit D-Q Transformation,” IEEE Transactions on Power Electronics 9, 1994, pp.390-396.

[68] T.M. Rowan and R.J. Kerkman, “A New Synchronous Current Regulator and an Analysis of Current Regulated PWM Inverters,” IEEE Transactions on Industrial Applications, IA-22, 1986, pp.678-690.

[69] D.N. Zmood, D.G. Holmes and D.H. Bode, "Frequency-domain Analysis of Three-phase Liner Current Regulators," IEEE Transactions Industrial Applications 37, 2001, pp.601-610.

[70] P. Feller, "Speed Control of an AC Motor by State Variables Feedback with Decoupling," Prceedings of IFAC on Control in Power Electronics and Electrical Drives, Lausanne, 1983, pp. 87-93.

[71] R.D. Lee, S.K. Sul and M.H. Park, "High Performance Current Regulator for a Field-oriented Induction Machine Controllers," IEEE Transactions Industrial Applications 30, 1994, pp. 1247-1257.

[72] J. Moerschel, "Signal Processor Based Field Oriented Vector Control for an Induction Motor Drive," Prceedings of EPE-Conference, Firenze, Italy, 1991, pp.2.145-2.150.

[73] D.G. Holmes and D.A. Martin, "Implementation of a Direct Digital Predictive Current Controller for Single and Three Phase Voltage Source Inverters," IEEE IAS 1996, San Diego, 1996, pp.906-913.

[74] T. Kawabata, T. Haneyoshi and R.G. Hoft, "Deadbeat Controlled PWM Inverter with Parameter Estimation Using only Voltage Sensor," IEEE PESC 1986, 1986, pp.576-583.

[75] J. Holtz and S. Stadtfeld, "A Predictive Controller for the Stator Current Vector of AC Machines Fed from a Switched Voltage Source," Prceedings of IPEC, Tokyo, 1983, pp. 1665-1675.

[76] A. Khambadkone and J. Holtz, "Low Switching Frequency High-power Inverter Drive Based on Field Oriented Pulse Width Modulation," Proceedings of EPE, 1991, pp. 4.672-4.677.

[77] M. Kheraluwala and D.M. Divan, "Delta Modulation Strategies for Resonant Link Inverters," Proceedings of IEEE PESC'87, 1987, pp.271-278.

[78] I. Boldea and S.A. Nasar, "Electric Drives," CRC Press, Boca Raton. FL, 1999.

[79] B.K. Bose, "Modern Power Electronics and AC Drives," Prentice-Hall, Englwood Cliffs, 2001.

[80] R. Krishnan, "Electric Motor Drives," Prentice-Hall,Englwood Cliffs, NJ, 2001.

[81] R. de Doncker and D.W. Novotny, "The Universal Field-Oriented Controller," IEEE Transactions on Industrial Applications, 30, 1994, pp. 92-100.

[82] M. Schiebe, S. Pferrer and R. Lauber, "Real-time Systems: Engineering and Applications," Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.

[83] J.A. Stankovic, "Real-time and embedded systems," *ACM Computing Surveys (CSUR),* Volume 28 , Issue 1, 1996, pp. 205-208.

[84] P. A. Laplante, "Real-time system design and analysis: an engineer's handbook," *IEEE Press: IEEE Computer Society Press*, New York, 2nd edition, 1997.

[85] A. Burns and A. Wellings, “Real-time Systems and Programming Languages”. Addison-Wesley, University of New York, 2nd edition, 1997.

[86] M. Lehman and L. Belady, “The Characteristics of Large Systems”. In Program Evolution – Processes of Software Change, APIC Studies and Data Processing No. 27, pp.289—329. Academic Press.

[87] Serlin, O. "Scheduling of Time Critical Processes," 925-932. *Proceedings of the Spring Joint Computer Conference*. Atlantic City, NJ, May 16-18, 1972. Montvale, NJ: American Federation of Information Processing Societies, 1972.

[88] Liu, C. L. & Layland, J. W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment." *Journal of the Association for Computing Machinery 20*, 1 (January 1973).

[89] Sha, Klein & Goodenough, J. "Rate Monotonic Analysis for Real-Time Systems," 129-155. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Boston, MA: Kluwer Academic Publishers, 1991.

[90] M. Shaw and D. Garlan, “Software Architecture: Perspectives on an Emerging Discipline,” Prentice-Hall, 1996.

[91] M. Pietrzak-David, B. de Fornel, A.M. Nogueira Lima and K. Jelassi, “Digital Control of an Induction Motor Drive by a Stochastic Estimator and Air Gap Magnetic Flux Feedback Loop,” IEEE Transactions on Power Electronics, vol. 7. no.2, April 1992, pp. 393 – 403.

[92] Y. Guo, H.C. Lee, X. Wang, B.-T. Ooi, “A Multiprocessor Digital Signal Processing System for Real-time Power Converter Applications,” Transactions on power systems, vol. 7, no. 2, May 1992, pp. 805-811.

[93] MathWorks SimuLink web site, http://www.mathworks.com/products/simulink/.

[94] MathWorks Real-Time Workshop web site, http://www.mathworks.com/products/rtw/.

[95] J. Guo, S.H. Edwards and D. Borojevich, “A Comparison of Power Electronics System Control Software Design Platforms: Dataflow Architecture and Mathworks Simulink + Real-Time Workshop Software Package,” CPES Seminar, Virginia Tech, Blacksburg. VA, 2003, pp.

[96] dSPACE website, http://www.dspaceinc.com/ww/en/inc/products.htm.

[97] A. Speck, “Reusable Industrial Control Systems,” IEEE Transactions on Industrial Electronics, Volume: 50 Issue: 3, June 2003, pp. 412 –418.

[98] “OSACA Open System Architecture for Controls Within Automation Systems,” EU, Brussels, Belgium, OSACA I&II Final Rep. Projects. EP6379&EP9115, Apr.1996.

[99] R.Bernhard, G.Schreck and C.Willnow, “A Cost Efficient Robot Control System—CEROS,” Prceedings of27th Int. Symposium of Industrial Robots, Robotics’96—Flexible Production Flexible Automation, Milan, Italy, 1996.

[100] ABB OPCoDe website, http://www.abb.com/.

[101] Lewis, R. W. (Robert W.), "Programming industrial control systems using IEC 1131-3," London : Institution of Electrical Engineers, 1998.

[102] A.L. Davis, and R. M. Keller, "Dataflow program graphs", IEEE Computer, vol. 15, no.2, Feb, 1982, pp.26-41.

[103] D. E. Culler, "Dataflow architectures," Annual Review of Computer Science, vol. 1, Annual Reviews Inc., Palo Alto, CA, 1986.

[104] B. S. Shuvra, P. K. Murthy, and E.A. Lee, "Software synthesis from dataflow graphs," "Kluwer Academic Publishers, Boston, 1996.

[105] K.Singh and S. H. Edwards, "DARK: Designing A High Performance Micro-kernel for Power Electronics Controllers," CPES Seminar, Virginia Tech, Blacksburg, VA, 2002, pp. 362-367.

[106] MathWorks Real-Time Workshop Embedded Coder web site, http://www.mathworks.com/products/rtwembedded.

[107] J. Francis, "A Synchronous Distributed Digital Control Architecture for High Power Converters," *M.S. Thesis*, Virginia Tech, 2004.

Appendix A. ECO EXAMPLES

A.1. Lookup_table

Lookup_table_Defs.h:

```
/*=============================================================*\
 |  Definitions of Lookup_table ECO
\*=============================================================*/
#ifndef LOOKUP_TABLE_DEFS_H
#define LOOKUP_TABLE_DEFS_H 1

typedef struct
{
    int Table_Length;
    int Step;
    float *Table;
    int Index;
    int Index_init;
} Lookup_table_Configuration;

/* -- Input Port Names --- */
#define LOOKUP_TABLE_START0
#define LOOKUP_TABLE_RESET 1

/* -- Output Port Names -- */
#define LOOKUP_TABLE_DATA  0

/*=============================================================*\
 |  Predefined Firing Masks
\*=============================================================*/
#define LOOKUP_TABLE_FIRING_MASK_DEFAULT            0x3
#define LOOKUP_TABLE_FIRING_MASK_RESET              0x2
#define     LOOKUP_TABLE_FIRING_MASK_INCREMENT      0x1
#define LOOKUP_TABLE_FIRING_MASK_EXCEPTION          0x4
```

```
/* -- Actions names -- */
#define LOOKUP_TABLE_DEFAULT_ACTION          0x0
#define LOOKUP_TABLE_RESET_ACTION            0x1
#define LOOKUP_TABLE_INCREMENT_ACTION        0X2
#define LOOKUP_TABLE_EXCEPTION_ACTION        0x3

#endif
```

Lookup_table.h

```
/*===========================================================*\
 | ECO: Lookup_table

\*===========================================================*/
# ifndef LOOKUP_TABLE_H
# define LOOKUP_TABLE_H 1

#include "Lookup_table_Defs.h"

/*--Definitions of input ports --*/
static Port_Descriptor Lookup_table_input_port_types[] =
{
      { Type_Bool,  0 }            /* Start */
    , { Type_Bool,  0}             /* Reset*/
    , { Type_Undefined,    0 }     /* Terminate list */
};

/*--Definitions of output ports--*/
static Port_Descriptor Lookup_table_output_port_types[] =
{
      { Type_Float,  0 }           /* Data          */
    , { Type_Undefined,    0 }     /* Terminate list          */
};
```

```
/* -- ECO Descriptor ----- */
void Lookup_table_Body( Process_Data* p );

static ECO_Descriptor Lookup_table =
{
        Lookup_table_Body
      , Lookup_table_input_port_types
      , Lookup_table_output_port_types
      , 500
      , 60
};
Firing_Mask Lookup_table_Mask[] = {
        LOOKUP_TABLE_FIRING_MASK_DEFAULT
      , LOOKUP_TABLE_FIRING_MASK_RESET
      , LOOKUP_TABLE_FIRING_MASK_INCREMENT
      , LOOKUP_TABLE_FIRING_MASK_EXCEPTION
};

/*==================================================================*\
 | Predefined Firing Rules
\*==================================================================*/
Priority_Firing_Mask Lookup_table_Firing_Rule_0[] =
{
        { LOOKUP_TABLE_FIRING_MASK_DEFAULT,     3 }
      , { LOOKUP_TABLE_FIRING_MASK_RESET,       4 }
      , { LOOKUP_TABLE_FIRING_MASK_INCREMENT, 5 }
      , { LOOKUP_TABLE_FIRING_MASK_EXCEPTION,  6 }
      , FIRING_MASK_TERMINATOR
};

Firing_Rule Lookup_table_Rule[] =
{
        Lookup_table_Firing_Rule_0
      , NULL
};


/*==================================================================*\
 | Lookup_table ECO
\*==================================================================*/
/*--------------------------------------------------------------------*/
# endif    /* LOOKUP_TABLE_H */
/*--------------------------------------------------------------------*/
```

Lookup_table.c

```
/*==================================================================*\
 | ECO: Lookup_table
 *--------------------------------------------------------------------*
 | This file provides the functionality of looking up sin table
\*==================================================================*/
#include "../os/ECO.h"
#include "Lookup_table_Defs.h"
#include "../os/log.h"

extern Process *current_process;
extern volatile Action_State actions_pending;/* To handle the time arrows and events    */
extern Context OS_env;                       /* Context saving space for the OS           */
extern Process **ready_queue;
extern Data_Channel *OS_All_DC;

extern ECO_Descriptor Lookup_table;
extern Firing_Mask Lookup_table_Mask[];
extern Firing_Rule Lookup_table_Rule[];

/*==================================================================*\
 | ECO Main Body
\*==================================================================*/
```

```
/*----------------------------------------------------------------------*\
 |  default_action( p )
 *----------------------------------------------------------------------*
 |  This operation carries out the "normal case" processing for this ECO.
\*----------------------------------------------------------------------*/
inline void Lookup_table_default_action(volatile Process_Data* p)
{
   float data;
   bool start, reset;

   Lookup_table_Configuration* config = (Lookup_table_Configuration*)(p->config);

   read_bool_DC(p->in_port[LOOKUP_TABLE_START],&start);
   read_bool_DC(p->in_port[LOOKUP_TABLE_RESET],&reset);

   //config = (Lookup_table_Configuration*)(p->config);

   if (reset)
   {
      config->Index = config->Index_init;
     config->Index += config->Step;
     if (config->Index >= config->Table_Length) config->Index = 0;
     data = config->Table[config->Index];
     write_float_DC(p->out_port[LOOKUP_TABLE_DATA], data);
   }
   else if (start)
   {
      config->Index += config->Step;
     if (config->Index >= config->Table_Length) config->Index = 0;
     data = config->Table[config->Index];
     write_float_DC(p->out_port[LOOKUP_TABLE_DATA], data);
   }
}
```

```
inline void Lookup_table_increment_action(volatile Process_Data* p)
{
    float data;

    bool start;

    Lookup_table_Configuration* config;

    read_bool_DC(p->in_port[LOOKUP_TABLE_START],&start);

    config = (Lookup_table_Configuration*)(p->config);

    data = config->Table[config->Index];

    config->Index += config->Step;

    if (config->Index >= config->Table_Length) config->Index = 0;

    write_float_DC(p->out_port[LOOKUP_TABLE_DATA], data);
}

inline void Lookup_table_reset_action(volatile Process_Data* p)
{
    float data;

    bool reset;

    Lookup_table_Configuration* config;

    read_bool_DC(p->in_port[LOOKUP_TABLE_RESET],&reset);

    config = (Lookup_table_Configuration*)(p->config);
```

```
       config->Index = config->Index_init;

       data = config->Table[config->Index];

       config->Index += config->Step;

       if (config->Index >= config->Table_Length) config->Index = 0;

       write_float_DC(p->out_port[LOOKUP_TABLE_DATA], data);
}

/*------------------------------------------------------------------------*\
 |  exception_handling( p )
 *------------------------------------------------------------------------*
 |  This operation carries out the "exception case" processing for this ECO.
\*------------------------------------------------------------------------*/

inline void Lookup_table_exception_handling(volatile Process_Data* p )
{
}

void Lookup_table_Body(volatile Process_Data* p )
{
   do {
      switch (p->wakeup_call)
      {
         case LOOKUP_TABLE_FIRING_MASK_RESET: Lookup_table_reset_action( p );
                break;
         case LOOKUP_TABLE_FIRING_MASK_INCREMENT:
                Lookup_table_increment_action ( p );
                break;
         case LOOKUP_TABLE_FIRING_MASK_DEFAULT:
                Lookup_table_default_action( p );
                break;
```

```
            case LOOKUP_TABLE_FIRING_MASK_EXCEPTION:
                    Lookup_table_exception_handling( p );
                    break;
            default:
                    //          Lookup_table_default_action( p );
                    /* ... some error flagged here ... */
                    break;
        }
    } while ( wait_to_fire( p ) );
}
```

A.2. PEBB_driver

PEBB_driver_Defs.h:

```
/*==================================================================*\
 |  PEBB_driver definitions
 \*==================================================================*/
#ifndef PEBB_DRIVER_DEFS_H
#define PEBB_DRIVER_DEFS_H 1

#include "../os/IO.h"

typedef struct
{
   In_Port_32 data_buffer;
} PEBB_driver_Configuration;

/* -- Input Port Names --- */
#define Duty_cycle  0

/* -- Firing Mask -- */
#define PEBB_DRIVER_FIRING_MASK_DEFAULT        0x1
#define PEBB_DRIVER_FIRING_MASK_EXCEPTION     0x2
```

```
/* -- Action Names -- */
#define PEBB_DRIVER_DEFAULT_ACTION       0x0
#define PEBB_DRIVER_EXCEPTION_ACTION   0x1
#endif
```

PEBB_driver.h:

```
/*==============================================================*\
 |  ECO: PEBB_driver

\*==============================================================*/
# ifndef PEBB_DRIVER_H
# define PEBB_DRIVER_H 1

//#include "ECO.h"
#include "PEBB_driver_Defs.h"

/*--Input ports definitions--*/
static Port_Descriptor PEBB_driver_input_port_types[] =
{
    { Type_Float,    0 }                    /* Duty_cycle               */
  , { Type_Undefined,        0 }        /* Terminate list */
};

/* -- ECO Descriptor ----- */
void PEBB_driver_Body( volatile Process_Data* p );

static ECO_Descriptor PEBB_driver =
{
    PEBB_driver_Body
  , PEBB_driver_input_port_types
  , NULL
  , 500
  , 70
};
```

```
/*==============================================================*\
 | Predefined Firing Masks
\*==============================================================*/

Firing_Mask PEBB_driver_Mask[] = {
    PEBB_DRIVER_FIRING_MASK_DEFAULT
    , PEBB_DRIVER_FIRING_MASK_EXCEPTION
};


/*==============================================================*\
 | Predefined Firing Rules
\*==============================================================*/

Priority_Firing_Mask PEBB_driver_Firing_Rule_0[] =
{
    { PEBB_DRIVER_FIRING_MASK_DEFAULT,          1 }
    , { PEBB_DRIVER_FIRING_MASK_EXCEPTION,      4 }
    , FIRING_MASK_TERMINATOR
};

Firing_Rule PEBB_driver_Rule[] =
{
    PEBB_driver_Firing_Rule_0
      , NULL
};

/*----------------------------------------------------------------------*/
# endif    /* PEBB_DRIVER_H */
/*----------------------------------------------------------------------*/
```

PEBB_driver.c:

```
/*==============================================================*\
 | ECO: PEBB_driver
 *-----------------------------------------------------------------------*
 | This file provides the implementation of the transformation from ABC
 | coordination to dqo coordination.
\*=====================================================================*/
#include "../os/ECO.h"
#include "../os/IO.h"
#include "PEBB_driver_Defs.h"
#include "../os/log.h"

extern Process *current_process;
extern volatile Action_State actions_pending;/* To handle the time arrows and events    */
extern Context OS_env;                  /* Context saving space for the OS             */
extern Process **ready_queue;
extern Data_Channel *OS_All_DC;



extern ECO_Descriptor PEBB_driver;
extern Firing_Mask PEBB_driver_Mask[];
extern Firing_Rule PEBB_driver_Rule[];

//For profiling
extern bool swcycle_finished;
extern OS_Process *OS_All_Process;

/*=====================================================================*\
 | ECO Main Body
\*=====================================================================*/

/* -- Declare helper functions. --------- */
/*-----------------------------------------------------------------------*\
 | default_action( p )
 *-----------------------------------------------------------------------*
 | This operation carries out the "normal case" processing for this ECO.
```

```
\*----------------------------------------------------------------------*/

inline void PEBB_driver_default_action( volatile Process_Data* p )
{
    /* Input variables */
    float duty_cycle;

    /* Intermediate variables */
    int clock_ticks;

    /* Hardware specifications */
    const int clock_cycle = 625;

    /* Configuration parameters */
  PEBB_driver_Configuration* config =
    (PEBB_driver_Configuration*)(p->config);

    /* Read input from data channels. */
    read_float_DC(p->in_port[Duty_cycle], &duty_cycle);

    /* Data format transfermation: duty cycle in float to clock ticks in integer */
    clock_ticks = (int)( duty_cycle * clock_cycle);

    *(int *)config->data_buffer = clock_ticks;
    //printf("Wake_up_call finished: %d\n",p->wakeup_call);
}


/*----------------------------------------------------------------------*\
 | exception_handling( p )
 *----------------------------------------------------------------------*
 | This operation carries out the "exception case" processing for this ECO.
\*----------------------------------------------------------------------*/
```

```
inline void PEBB_driver_exception_handling( volatile Process_Data* p )
{
}

void PEBB_driver_Body( volatile Process_Data* p )
{
    //printf("Reached process PEBB : %d\n",p->current_priority);
    do
   {
    log_1( "PEBB_driver_Body ( %x ): starting -----------------------------\n\n", p );
            switch (p->wakeup_call)
            {
              case PEBB_DRIVER_FIRING_MASK_DEFAULT:
                    PEBB_driver_default_action( p );
                    break;
              case PEBB_DRIVER_FIRING_MASK_EXCEPTION:
                    PEBB_driver_exception_handling( p );
                    break;
              default:
                    //PEBB_driver_default_action( p );
                    /* ... some error flagged here ... */
                 break;
            }
    log_1( "PEBB_driver_Body ( %x ): Going to sleep -----------------------------\n\n\n", p );

   } while ( wait_to_fire( p ) );
}
```

Appendix B. DATAFLOW GRAPH EXAMPLE: CLOSED-LOOP 3-PHASE VSI

```
/*=================================================================*\
 | Closed-loop VSI
 *-----------------------------------------------------------------*
 | This file shows the basic structure of an ECO application built from
 | library ECOs.  First, it defines the startup configuration parameters
 | for all the ECO instances in the application.  Second, it defines the
 | ECO instances as a set of DFG nodes.  Third, is specifies the DFG edges
 | (data channels) that connect the ECO instances, along with their
 | properties.
 |
 | The goal is to be able to specify all of the application-specific
 | information here (assuming it can be built entirely from library ECOs).
 | Simply compiling and linking this application description with the ECO
 | library and the DARK kernel is all that is necessary to generate the
 | application.
\*=================================================================*/

#define APPL 1

#include "../OS/ECO.h"
#include "IO_Appl.h"
#include "ECO_Library.h"
#include "sin_table.h"
#include "cos_table.h"

/*=================================================================*\
 | Configuration Parameters
 *-----------------------------------------------------------------*
 | The Set of Nodes is defined in the following section.  This section
 | defines the individual configuration parameters for each node in the DFG.
```

```
\*============================================================*/

Lookup_table_Configuration Node_1_Configuration =
{
    166             /* sin table length */
    , 1             /* step                          */
  , SIN_TABLE     /* parameter names goes here.   */
    , 0
    , 0
};

Lookup_table_Configuration Node_2_Configuration =
{
    166             /* sin table length */
    , 1             /* step                          */
  , COS_TABLE     /* parameter names goes here.   */
    , 0
    , 0
};

Regulator_Configuration Node_6_Configuration =
{
      0.034
     , 0.866
     , 2.27e-3
     , 0.034
     , 0.866
     , 2.27e-3
};


/*============================================================*\
 | Node Set
\*============================================================*/
```

```
DFG_Node DFG_Node_Set[] =
{
        {
                  0
        }
      // node 1: lookup table (sin)
      , {
          &Lookup_table
        , &Node_1_Configuration
        , Lookup_table_Rule
        , 0
        , 8
        , LOOKUP_TABLE_FIRING_MASK_INCREMENT
        , 0
        }

    // node 2: lookup table (cos)
      , {
          &Lookup_table
        , &Node_2_Configuration
        , Lookup_table_Rule
        , 0
        , 7
        , LOOKUP_TABLE_FIRING_MASK_INCREMENT
        , 0
        }

      // node 3: duplicator (for sin)
      , {
          &duplicator
        , 0
        , duplicator_Rule
        , 0
        , 6
```

```
  , DUPLICATOR_FIRING_MASK_DEFAULT
  , 0
  }

// node 4: duplicator (for cos)
, {
    &duplicator
  , 0
  , duplicator_Rule
  , 0
  , 5
  , DUPLICATOR_FIRING_MASK_DEFAULT
  , 0
  }

// node 5: abcln_dq transformation
, {
    &abcln_dq
  , 0
  , abcln_dq_Rule
  , 0
  , 4
  , ABCLN_DQ_FIRING_MASK_DEFAULT
  , 0
  }

// node 6: regulator (voltage)
, {
    &Regulator
  , &Node_6_Configuration
  , Regulator_Rule
  , 0
  , 3
  , REGULATOR_FIRING_MASK_DEFAULT
```

```
          , 0
          }

        // node 7: dqo_albe transformation
        , {
            &dq_albe
          , 0
          , dq_albe_Rule
          , 0
          , 2
          , DQ_ALBE_FIRING_MASK_DEFAULT
          , 0
          }

        // node 8: modulator (SVM)
        , {
            &Modulator
          , 0
          , Modulator_Rule
          , 0
          , 1
          , MODULATOR_FIRING_MASK_DEFAULT
          , 0
          }
        , NODE_TERMINATOR
};


/*====================================================================*\
 |  Edge Set
\*====================================================================*/
DFG_Edge DFG_Edge_Set[] =
{
        // Edge 0: va
```

```
    {
      Type_Float
     , 0, 2           // Source name, port name.
     , 5, 0           // Sink name, port name.
     , OS_Block
     , 3                       // Stores 5 elements.
     , 0                       // Not an array.
     , true, 0                 // *Not* It is interrupt-driven.
     , 1                       // Update frequency
     }

    // Edge 1: vb
   , {
      Type_Float
     , 0, 3           // Source name, port name.
     , 5, 1           // Sink name, port name.
     , OS_Block
     , 3                       // Stores 5 elements.
     , 0                       // Not an array.
     , true, 0                 // *Not* It is interrupt-driven.
     , 1                       // Update frequency
     }

     // Edge 2: start (lookup sin table)
   , {
      Type_Bool
     , 0, 0           // Source name, port name.
     , 1, 0           // Sink name, port name.
     , OS_Block
     , 3                       // Stores 5 elements.
     , 0                       // Not an array.
     , true, 0                 // *Not* It is interrupt-driven.
     , 1                       // Update frequency
     }
```

```
  // Edge 3: start (lookup cos table)
, {
    Type_Bool
  , 0, 1          // Source name, port name.
  , 2, 0          // Sink name, port name.
  , OS_Block
  , 3                   // Stores 5 elements.
  , 0                   // Not an array.
  , true, 0       // *Not* It is interrupt-driven.
  , 1                   // Update frequency
  }

  // Edge 4: sin(theta)
, {
    Type_Float
  , 1, 0          // Source name, port name.
  , 3, 0          // Sink name, port name.
  , OS_Block
  , 3                   // Stores 5 elements.
  , 0                   // Not an array.
  , false, 0      // *Not* It is interrupt-driven.
  , 1                   // Update frequency
  }

  // Edge 5: cos(theta)
, {
    Type_Float
  , 2, 0          // Source name, port name.
  , 4, 0          // Sink name, port name.
  , OS_Block
  , 3                   // Stores 5 elements.
  , 0                   // Not an array.
  , false, 0      // Not interrupt-driven.
```

```
   , 1                         // Update frequency
   }

  // Edge 6: sin (duplicate 1)
 , {
    Type_Float
   , 3, 0              // Source name, port name.
   , 5, 3              // Sink name, port name.
   , OS_Block
   , 3                         // Stores 5 elements.
   , 0                         // Not an array.
   , false, 0          // Not interrupt-driven.
   , 1                         // Update frequency
   }

   // Edge 7: sin(duplicate 2)
 , {
    Type_Float
   , 3, 1              // Source name, port name.
   , 7, 2              // Sink name, port name.
   , OS_Block
   , 3                         // Stores 5 elements.
   , 0                         // Not an array.
   , false, 0          // Not interrupt-driven.
   , 1                         // Update frequency
   }

   // Edge 8: cos(duplicate 1)
 , {
    Type_Float
   , 4, 0              // Source name, port name.
   , 5, 4              // Sink name, port name.
   , OS_Block
   , 3                         // Stores 5 elements.
   , 0                   // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                   // Update frequency
   }

   // Edge 9: cos(duplicate 2)
 , {
    Type_Float
   , 4, 1          // Source name, port name.
   , 7, 3          // Sink name, port name.
   , OS_Block
   , 3                   // Stores 5 elements.
   , 0                   // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                   // Update frequency
   }

   // Edge 10: vd_ref
 , {
    Type_Float
   , 0, 4          // Source name, port name.
   , 6, 2          // Sink name, port name.
   , OS_Block
   , 3                   // Stores 5 elements.
   , 0                   // Not an array.
   , true, 0       // Not interrupt-driven.
   , 1                   // Update frequency
   }

   // Edge 11: vq_ref
 , {
    Type_Float
   , 0, 5          // Source name, port name.
   , 6, 3          // Sink name, port name.
   , OS_Block
   , 3                  // Stores 5 elements.
   , 0                  // Not an array.
   , true, 0       // Not interrupt-driven.
   , 1                  // Update frequency
   }

   // Edge 12: vd
 , {
    Type_Float
   , 5, 0          // Source name, port name.
   , 6, 0          // Sink name, port name.
   , OS_Block
   , 3                  // Stores 5 elements.
   , 0                  // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                  // Update frequency
   }

   // Edge 13: vq
 , {
    Type_Float
   , 5, 1          // Source name, port name.
   , 6, 1          // Sink name, port name.
   , OS_Block
   , 3                  // Stores 5 elements.
   , 0                  // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                  // Update frequency
   }

   // Edge 14: dd
 , {
    Type_Float
   , 6, 0          // Source name, port name.
   , 7, 0          // Sink name, port name.
   , OS_Block
   , 3                    // Stores 5 elements.
   , 0                    // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                    // Update frequency
   }

   // Edge 15: dq
 , {
    Type_Float
   , 6, 1          // Source name, port name.
   , 7, 1          // Sink name, port name.
   , OS_Block
   , 3                    // Stores 5 elements.
   , 0                    // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                    // Update frequency
   }

   // Edge 16: d_alpha
 , {
    Type_Float
   , 7, 0          // Source name, port name.
   , 8, 0          // Sink name, port name.
   , OS_Block
   , 3                    // Stores 5 elements.
   , 0                    // Not an array.
   , false, 0      // Not interrupt-driven.
   , 1                    // Update frequency
   }

   // Edge 17: d_beta
```

```
    , {
       Type_Float
      , 7, 1           // Source name, port name.
      , 8, 1           // Sink name, port name.
      , OS_Block
      , 3                        // Stores 5 elements.
      , 0                        // Not an array.
      , false, 0       // Not interrupt-driven.
      , 1                        // Update frequency
      }

      // Edge 18: vc
      , {
       Type_Float
      , 0, 6           // Source name, port name.
      , 5, 2           // Sink name, port name.
      , OS_Block
      , 3                        // Stores 5 elements.
      , 0                        // Not an array.
      , true, 0        // Not interrupt-driven.
      , 1                        // Update frequency
      }

      , EDGE_TERMINATOR
};
/*----------------------------------------------------------------------*/
```

VITA

The author, Jinghong Guo, was born in Nanjing, Jiangsu, China on July 23rd, 1973. She received B.S. degree and M.S. degree both in Computer Science and Engineering from Nanjing University of Aeronautics and Astronautics, China in 1995 and 1998, respectively. In 1999, she joined the Center for Power Electronics Systems (CPES) of Virginia Polytechnic Institute and State University, Blacksburg, VA, for her Ph.D. program. Her research interests include digital control design, digital communications network design for medium- to high- power conversion systems, and power management.