Designing Reusable, Reconfigurable Control Software for Power Electronics Systems

Jinghong Guo, Stephen Edwards and Dushan Borojevic

Center for Power Electronics Systems

The Bradley Department of Electrical and Computer Engineering

Virginia Polytechnic Institute and State University

Blacksburg, VA 24061 USA

Abstract—**This paper presents dataflow architecture to design reusable and reconfigurable control software of power electronics systems. Reasons that traditional "main-program-and-subroutine" architecture causes lots of software issues of power electronics control software are analyzed. The infrastructure of dataflow architecture will be discussed. And a 3-phase inverter application is used as an example to evaluate the proposed software architecture vs. the legacy "main-program-and-subroutine" approach.**

I. INTRODUCTION

A power electronics control system is a real-time system—it operates using limited resources and under a set of deadlines. As distributed control architectures become commonplace for power electronics systems, the size and complexity of the corresponding control software will increase. As a result, the design and specification of the overall software structure become more significant issues than the choice of algorithms and data structures used in the computation. Structural issues in software design 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.

The traditional procedural or imperative approach to designing embedded control software results in a main-program-and-subroutine architecture that has several disadvantages. The control software is harder to maintain and modify. The software is tightly coupled to the hardware. New systems typically require significant redesign effort, because the main-program-and-subroutine architecture does not support software reusability well.

To address these shortcomings, a different approach to structuring software designs is presented. Dataflow is a style of software architecture that strongly supports reusability and reconfigurability. Based on the specifications and requirements of power electronics control software, the dataflow architecture style is compared with other candidate software architectures to illustrate its benefits for building control software.

In the dataflow style, a control application is implemented as a set of concurrently executing processes, which we call elementary control objects (ECOs). 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, producing 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.

The support infrastructure needed to write dataflow-based control algorithms is described. An application example will be used to show how control software constructed can be easily constructed from a library of reusable parts using this strategy, as well as how the architecture works in practice.

The performance of dataflow-based control applications is compared with similar applications using a more conventional main-program-and-subroutine architecture. The results show that using dataflow could save engineering effort while preserving the quality of the software system. The greatest drawback to using a dataflow design is that dataflow implementations typically introduce more performance overhead than traditional approaches. Bottlenecks in dataflow architectures will be analyzed and possible solutions will be presented.

II. IMPROVEMENTS IN POWER ELECTRONICS SYSTEMS HARDWARE ARCHITECTURES

Fig. 1 shows a traditional structure for the hardware of a digitally controlled power electronics system, which features centralized control and centralized power stage management. The design can be customized to meet application requirements, but it lacks standardization and modularization, which leads to duplication of effort in system development, additional complexity in maintenance, and higher cost.

This work was supported primarily by the ERC Program of the National Science Foundation under Award Number EEC-9731677.

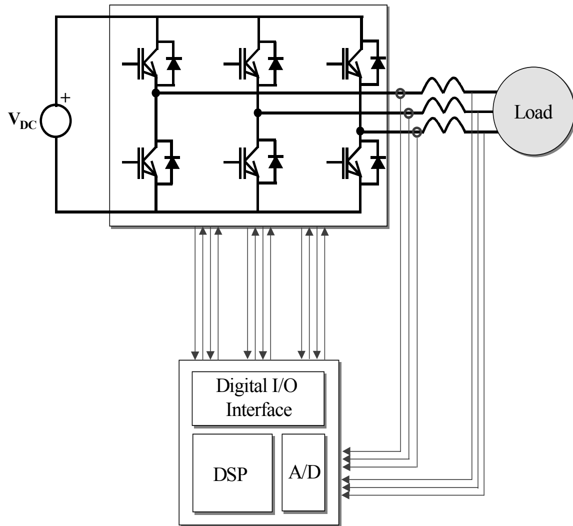

Fig. 1 Centralized approach to power electronics system hardware design.

The power electronics building block (PEBB) concept was proposed for modularize power stages [2, 3]. The application manager/hardware manager (AM/HM) architecture was then proposed to divide the control into several layers [4]. Fig. 2 shows the hierarchical structure of PEBB-based power electronics system. Standardizing interfaces between levels increases modularity while also supporting additional flexibility in system design and construction [5].

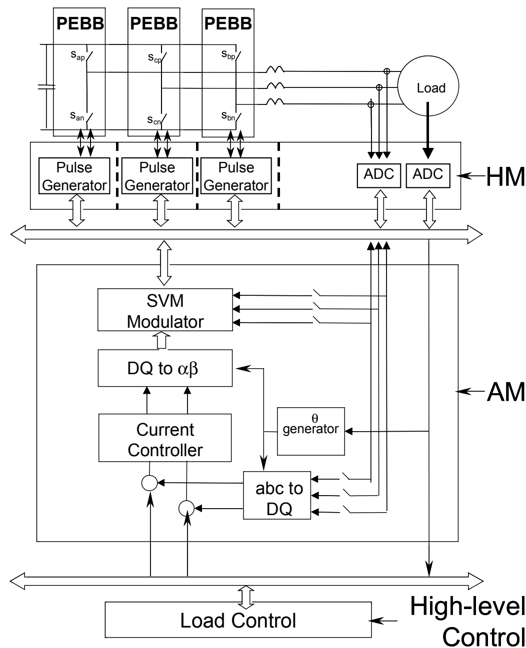

Fig. 2 Hierarchical power electronics system hardware structure.

The traditional main-program-and-subroutine software design approach is to software architecture what the older, centralized hardware design of Fig. 1 is to hardware architecture. It leads to a variety of software problems that only become more severe in hierarchical power electronics system designs. In hierarchical systems, besides implementing the control algorithm, the control software also needs to handle communication and coordination between distributed hardware elements. This adds more complexity to the control software design.

Generally speaking, there are three types of data related to the real-time control software, input data, output data and calibration constants or algorithm coefficients. The real-time control software takes physical feedback from power stage as input, and generates control information as output to adjust behavior of power stage, as shown in Fig. 3. Management of each type of data is associated with several kinds of information. Some data items are directly related to hardware specifications, some are decided by the software architecture style, and others may depend on application requirements. However, under the main-program-and-subroutine architecture, different kinds of information are not explicitly identified or separated, and tend to be coded in the same way. This results in control software that is tightly bound to the specific hardware and application; all types of information are hard coded and the interaction mechanisms used within the architecture strongly couple the components (subroutines), limiting their reusability.

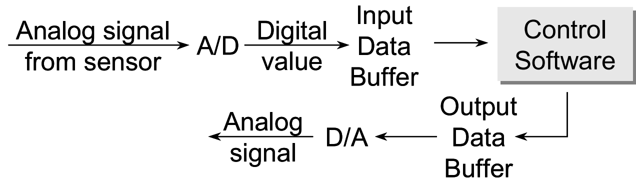

Fig. 3 Real-time control procedure in power electronics systems.

III. AN ARCHITECTURE FOR REUSABLE CONTROL SOFTWARE IN POWER ELECTRONICS SYSTEMS

The choice of software architecture has a dramatic impact on the issues discussed in Section II. This paper proposes alternative software architecture— dataflow architecture—for developing control software configured from reusable modules. The proposed software architecture has the following properties: minimal coupling between software components; encapsulated hardware dependencies; highly reusable components; component and architectural reconfigurability; transparent support for distributed execution; scalability; expandability; and upgradeability.

A. An Outline of the Dataflow Architecture

Software written using a 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 Fig. 4. A node is a computational component, and an arrow is a buffered data channel. A control algorithm is divided into nodes first.

Each concurrently executing node is a self-contained software part with well-defined functionality. Data channels provide the sole mechanism by which nodes can interact and communicate with each other, ensuring lower 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 data flow components. To best capitalize on the reusability features of the dataflow architecture style, the design of a library of standardized software components will be discussed. Based on the design library, a new system can be rapidly configured from an existing collection of components.

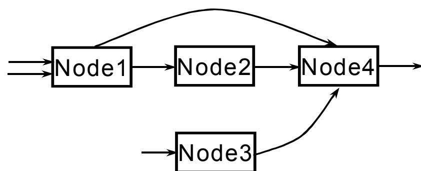

Fig. 4 Dataflow architecture.

B. Elementary Control Objects (ECOs)

Data flow nodes in this architectural style are called elementary control objects (ECOs). Each ECO manipulates input data that it receives according to its functionality, generating output that can be connected to other ECOs. There are no explicit calls between ECOs—in fact, no ECO knows anything about the other nodes the system comprises, or the identities of the other nodes with which it communicates. ECOs are naturally independent, so they are naturally able to execute concurrently. Thus, distributed control of power electronics system is easy to build with ECOs. An ECO contains an input and output data channel description, a startup parameter description, and an implementation, as illustrated in Fig. 5.

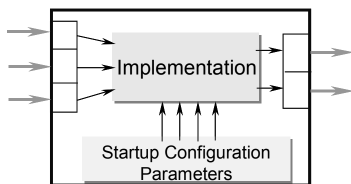

Fig. 5 ECO structure.

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. Fig. 6 shows several examples of computational ECOs. A coordination ECO, on the other hand, is designed to support transparent management and control of distributed system hardware assets, as exemplified in Figure 7. Driver ECOs encapsulate hardware dependencies and provide a standard program interface to control hardware. Figure 8 shows an A/D driver ECO.

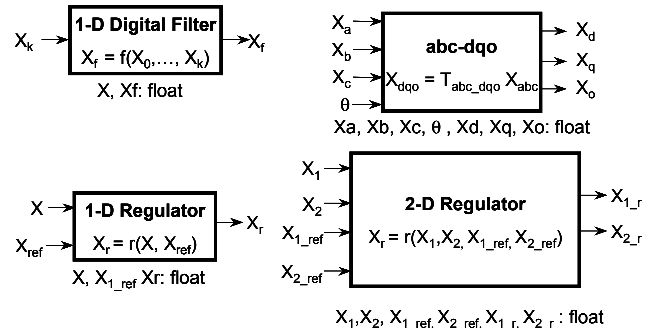

Fig. 6 Examples of computational ECO.

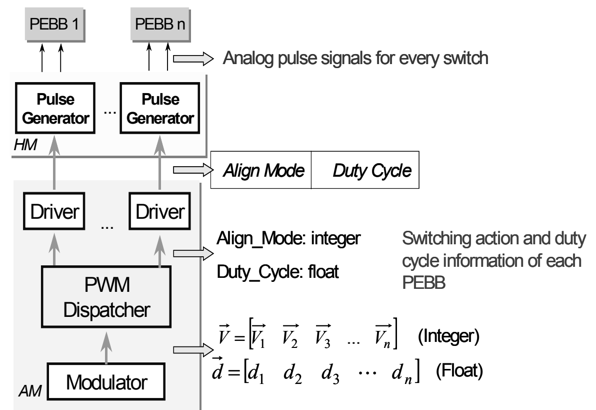

Fig. 7 Example of coordination.

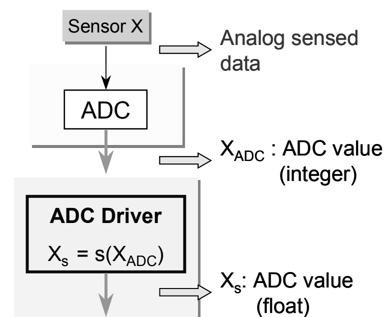

Fig. 8 A/D driver ECO.

D. Data Channels

Data channels serve as the sole communication paths connecting ECOs into a 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 as 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 detect certain kinds of interconnection errors in an application early during development, rather than later during operational testing. Each data channel has a data queue to buffer data between ECOs operating at different speeds. A data channel's data type, buffer size and source and sink connections are all configured by the application designer when developing the overall software structure.

E. The Dataflow Graph

The dataflow graph describes the control software configuration as a composition of ECOs interconnected with data channels. 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. Fig. 9 shows two different partitioning decisions for allocating dataflow graph elements to two separate hardware resources.

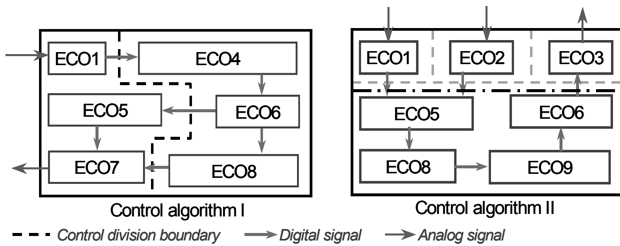

Fig. 9 Dataflow graph examples.

F. Dataflow Architecture Real-time Kernel (DARK)

The dataflow architecture infrastructure for ECO-based control software amounts to a small embedded operating system for ECOs. As a result, this infrastructure requires a real-time kernel –DARK—that supports lightweight process management, data channel management, system resource allocation, device driver support, and interrupt handling support. DARK is scalable to meet real-time requirements of a specific application. More DARK features involved may impose more overheads to the application performance.

IV. EVALUATION OF THE DATAFLOW ARCHITECTURE AND ECO LIBRARY APPROACH

The on-going evaluation efforts for this approach aim to show the decrease in control software development complexity, better support for distributed hardware, and increase in reusability and reconfigurability gained by use of this architectural style. The imposed performance overhead of the dataflow approach is also assessed through the evaluation experiments.

Evaluation activities so far have been based on a PEBB-based three-phase inverter application. The system hardware setup is shown in Fig. 10.

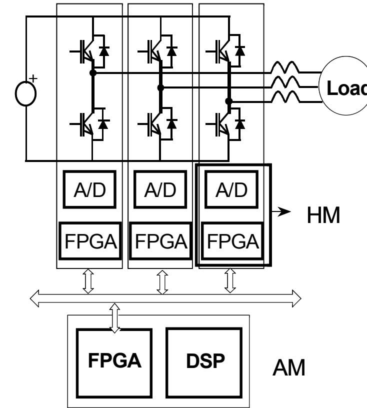

Fig. 10 PEBB-based three-phase inverter system.

Each phase leg is constructed as a PEBB. The control is implemented with Application Manager/Hardware manager structure. The supply DC voltage is 60V, and the switching frequency is 20KHz. Several experiments are designed to compare the control software development complexity and performance of the proposed approach with the existing control software of the main-program-and-subroutine approach. Two control applications are implemented on this hardware setup.

Application 1: Open loop control.

Application 2: Current close loop control.

Two software architecture approaches are used to compose control software.

Approach 1: Main-program-and-subroutine architecture;

Approach 2: Dataflow architecture.

All code of the main-program-and-subroutine version is written in Assembly language. All code in the dataflow version is written in C programming language.

The digital signal processor (DSP) used has 40MHz instruction speed and 64K words internal memory.

A. Experiment 1: Open loop control application

The open loop control application is implemented in both software approaches mentioned above.

The main-program-and-subroutine version is scratched from the assembly instruction level. For the software can communicate with hardware, the software designer has to know hardware distribution and interface in detail. To implement the dataflow version, what need to do is map the open loop control algorithm into a dataflow graph, as shown in Fig. 11. All the computational ECOs are library ECOs. All driver ECOs are supposed to be provided by hardware designers. The coordination ECO is provided by the network designer. What left for the software designer is to describe the selection and connection of suitable ECOs in a dataflow graph. The first run the dataflow version is on the full-featured DARK.

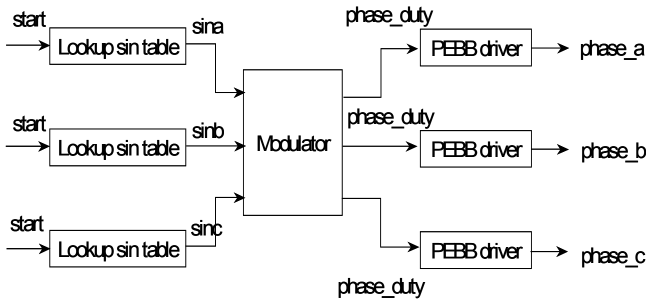

Fig. 11 Dataflow graph of open loop control for PEBB-based 3-phase inverter.

The real-time performances of both versions are analyzed in simulator. Tab. 1 shows the performance comparison of the main-program-and-subroutine and dataflow versions of the open loop control application.

Software Approach	Running time for one switching period (instruction cycles)	Memory (Word)
Main-program-and-subroutine	50	0.9K
Dataflow Architecture	8739	11.4K

Tab. 1 Performance comparison for 3-phase inverter open loop control.

Tab. 1 shows significant performance overhead imposed by the dataflow approach. For the 40MHz DSP used for the experiment, the computation bandwidth of a 3-phase inverter with 20KHz switching frequency is 2000 instruction cycles per switching period. The dataflow version with the full-featured DARK is not able to meet this real-time deadline. Statistics from profiling tools in the simulator shows that DARK takes around 95% of the running time, as shown in Fig. 12. Hence, several efforts of how to optimize the dataflow code and scale down DARK to meet real-time deadline are discussed.

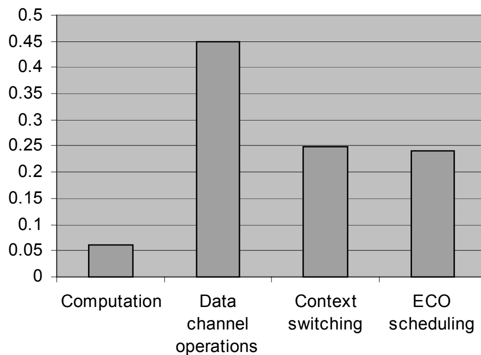

Fig. 12 Performance analysis of dataflow version control software

Optimization effort 1: inline functions

One of the main sources of dataflow code overhead is programming language. Most part of the dataflow version is written in C, which has lower code efficiency than Assembly language. For example, a normal C function call takes 20 instruction cycles on average, compared to 3 instruction cycles for an Assembly function call. By turning normal C function calls into inline functions, the running time of the dataflow version is cut down by 12%, as shown in Tab. 2. The tradeoff is more memory usage. However, the bottleneck of the dataflow architecture performance is running time.

Optimization and simplification efforts	Running time for one switching period (instruction cycles)	Memory (Word)
Effort 1	7687	10.9K
Effort 2	4196	11.3K
Effort 3	1866	10.2K

Tab. 2 Performance analysis of optimization of dataflow version.

Optimization effort 2: macros

Because the code size limit on inline function by the compiler, a large percent of normal C function cannot be inlined as desired. Macros do not have such code size limitation. By turning C functions into macros, the running time is reduced around 52%, as shown in Tab. 2.

Optimization effort 3: scaling down DARK

The full-featured DARK is capable of pre-emptive ECO process scheduling and data channel queue maintenance. In the preemptive ECO process scheduling, a running ECO process can be pre-empted by another ECO process that has higher priority. The pre-emptive scheduling helps the system to response more rapidly to more important events. However, because preemption could happens at any time, the context switching from one ECO process to another ECO process needs to save the whole running environment of the running process and restore the running environment of the scheduled process, which is time consuming. As shown in Figure 13, context switching takes around 25% of the running time. If DARK is scaled down to non-preemptive scheduling, a running ECO process will not be interrupted by any other ECO process until it finishes it task. Hence, the context switching between ECO processes can be minimized. Within a control loop, the correct execution sequence of ECO processes is ensured by data channel connections and firing rules. Preemptive scheduling takes more unnecessary running time in this open-loop control application. For applications that have multi control loops, preemptive scheduling should be applied between control loops.

Fig. 12 also indicates that data channel operations take significant running time. The full-featured DARK is able to handle a queue for each data channel. The purpose of data channel queue is to buffer data between source ECO process and sink ECO process with different data update speed. For ECO processes that are in the same control loop and allocated to the same processor, normally they have the same data update speed. Under this circumstance, the data channel

queue can be substituted by a data unit, which greatly simplifies the data channel operations.

By scaling down DARK, the running time of dataflow version is cut down to 1866 instruction cycles for one switching period, which meets the real-time deadline of the 3-phase inverter open loop control.

Experiment 2: Close loop 3-phase inverter control application

A current close loop control application is implemented on the same hardware using both software approaches. The main-program-and-subroutine version of the close loop control application is again scratched from the assembly instruction level. There is little can be carried on from the open loop control application. The main work to implement the dataflow version is to generate a dataflow graph that describes the close loop control application, as shown in Figure 9.

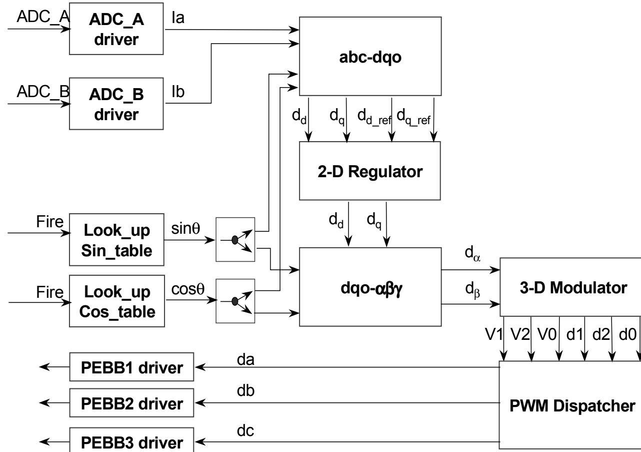

Fig. 13 Dataflow graph of current close loop control for 3-phase inverter.

The real-time performances of both versions are analyzed in simulator. Tab. 3 shows the performance comparison of the main-program-and-subroutine and dataflow versions of the close loop control application. The dataflow version runs on the simplified DARK.

Software Approach	**Running time for one switching period (instruction cycles)**	**Memory (Word)**
Main-program-and-subroutine	1106	2.1K
Dataflow Architecture	3296	18.6K

Tab. 3 Performance comparison for 3-phase inverter close loop control.

V. Conclusion and Future Work

The dataflow architecture provides a framework that supports software modularity, reusability, and reconfigurability. It reduces the overall software development cost and time by decreasing the complexity of development and testing. This approach also reduces the required redesign effort when building a new system. Under the dataflow architecture, problem domains are separated, so that engineers with different expertise can concentrate on their specialty. Systems built under the proposed architecture are easy to expand and upgrade.

The main disadvantage of the dataflow architecture is the performance overhead introduced into the system. Because software components are independent of each other, support software is needed to handle coordination and communication between components. This disadvantage can often be addressed by using fast controller hardware [6], which can provide better results than pouring additional resources into the software development effort. In addition, the use of ECOs from a library may make it more difficult to readily incorporate application-specific optimizations to processing and control algorithms.

References

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

[2] J.D. Van Wyk and F.C. Lee, "Power Electronics Technology at the Dawn of the New Millennium—Status and Future," *IEEE PESC Conference Proceedings,* 1999, vol. 1, pp. 3–12.

[3] A.A. Jaecklin, "Future Devices and Modules for Power Electronic Applications," *European Conference on Power Electronics and Applications, EPE,* September 1999.

[4] I. Celanovic, I. Milosavljevic, D. Boroyevich, J. Guo, and R. Cooley, "A New Distributed Controller for the Next Generation Power Electronics Building Blocks," *15th Annual IEEE Applied Power Electronics Conference Proceedings,* February 2000, pp. 889–894.

[5] J. Guo, D. Borojevic, and I. Celanovic, "Software Structure of the PEBB-based Plug and Play Power Electronics Systems," *16th IEEE Applied Power Electronics Conference and Exposition,* Anaheim, Ca, 2000.

[6] J. Francis and D. Borojevic, "Design of a Universal Controller for Distributed Control and Power Electronics Applications," *CPES Seminar,* 2001.