Implementing Dataflow-based Control Software for Power Electronics Systems

Jinghong Guo, Stephen H. 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

*Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061 USA

***Abstract*—Dataflow provides a natural way to describe the structure of many control algorithms. This paper presents a method for implementing modular, reusable and reconfigurable control software for power electronics systems using a dataflow architecture. A 3-phase inverter application is used as an example to illustrate the approach and evaluate the proposed software vs. the legacy "main-program-and-subroutine" approach.**

I. INTRODUCTION

A "main-program-and-subroutine" software architecture typifies the traditional procedural approach to designing embedded control software for power electronics systems. Such a software design strategy 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 on the software side, because the main-program-and-subroutine architecture does not support software reusability well. As power electronics systems get bigger and complicated, those disadvantages becomes bottlenecks in control software design.

A different approach to structuring software designs has been proposed to address these shortcomings [1]. Dataflow is a style of software architecture [2] that strongly supports reusability and reconfigurability. It 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. This paper focuses on how to implement all those properties of dataflow software in power electronics systems.

This paper illustrates how one can implement dataflow-oriented control software in terms of elementary control objects (ECOs)—the data triggered computational processes that perform actions—and data channels—buffered message pipes between ECOs that support inter-component communication. A closed loop 3-phase inverter controller will be used as an application to illustrate the approach. When the system hardware changes, this control software can be reconfigured easily and naturally to conform to the changes. The dataflow-based control software described here has been tested on a real 3-phase inverter. The performance will be compared to that of the control software written in the traditional main-program-subroutine style.

Section II presents a brief introduction to the dataflow approach, and Section III introduces the PEBB-based 3-phase inverter as an application example. In Section IV, the internal structure of dataflow processes—ECOs—is shown through an easy understood ECO example. The design of data channels is also discussed. Section V describes the performance of the dataflow architectural control software for the PEBB-based 3-phase inverter and compares it with the traditional main-program-and-subroutine approach.

II. INFRASTRUCTURE FOR DATAFLOW COMPUTING

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. Here, a component is called elementary control object (ECO). A control algorithm is divided into ECOs 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. By using a design library of commonly needed ECOs, a new system can be rapidly configured from an existing collection of components.

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 a power electronics system is easy to build with ECOs. 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. A coordination ECO, on the other hand, is designed to support transparent management and control of distributed system hardware assets. Driver ECOs encapsulate hardware dependencies and provide a standard program interface to control hardware. .

This work was supported primarily by the Office of Naval Research under Award Number N00015-01-1-0954 and the ERC Program of the National Science Foundation under Award Number EEC-9731677.

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.

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.

To support execution of the ECO-based dataflow applications, a custom, embeddable real-time operating system (RTOS) was created. The Dataflow Architecture Real-time Kernel (DARK) [5] provides support for lightweight process management, data channel management, system resource allocation, device driver support, and interrupt handling.

III. DATAFLOW-BASED CONTROL SOFTWARE FOR A CLOSED LOOP 3-PHASE INVERTER

In this section, the control of a PEBB-base closed loop 3-phase inverter is used as an example to show how control software can be constructed in dataflow architecture. The switching frequency of the inverter is 20KHz.

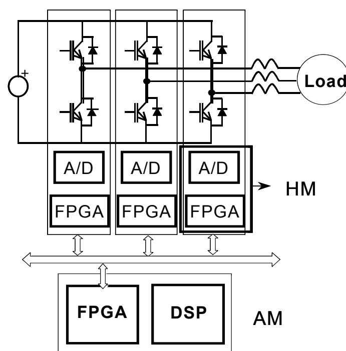

Fig. 1. PEBB-based 3-phase inverter system with AM-HM distributed control.

The system structure of the PEBB-based 3-phase inverter with AM-HM control structure [3], [4] is shown in Figure 1. Application manager (AM) and hardware managers (HM) conforms the distributed control of the system. The AM performs most the computation of the inverter control algorithm, such as PWM information for each phase leg, while HMs correspond to translate AM control information to local control commands and fast protection. The dataflow graph of the control algorithm in AM is shown in Figure 2.

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" 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 to get duty cycles 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.

If the inverter system is reconstructed as shown in Figure 3, the distributed control substituted by a centralized control, there are minor changes need to be done to the control software designed for the system shown in Figure 3. The PWM Dispatcher ECO is no longer needed because there is no hardware distribution in the power stage. And PEBB drivers will be eliminated from the dataflow graph. All the changes are made in the dataflow graph. And the rest of the control software will be the same.

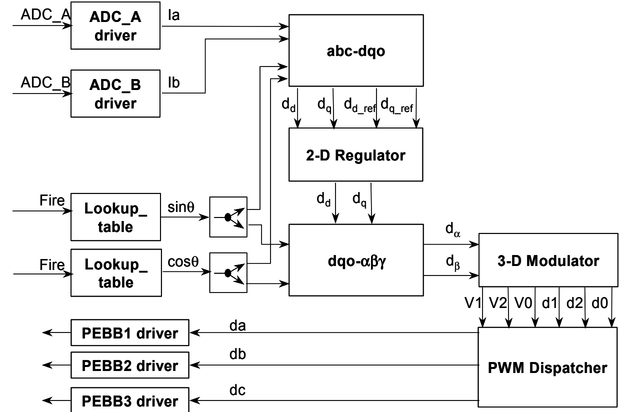

Fig. 2. Dataflow graph of PEBB-based 3-phase closed loop inverter control.

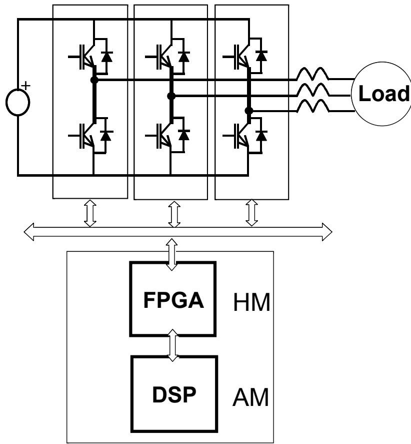

Fig. 3. PEBB-based 3-phase inverter with centralized control.

IV. Implementation of Dataflow Architecture

Achieving rapid reconfigurability and high reuse while maintaining code quality requires one to use structural programming language features to the greatest extent possible. On the other hand, the control software of power electronics, normally running in a digital signal processor (DSP) on a digital controller board, has tight real time dead lines to meet. To maximize code efficiency, the proposed dataflow software is written in C combined with DSP assembly language.

A. ECO Design

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. Figure 4 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 Figure 5.

```
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;
    }
  }
}
```

Fig. 4. Pseudo-code for a typical ECO implementation.

```
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;
```

Fig. 5. Process data structure.

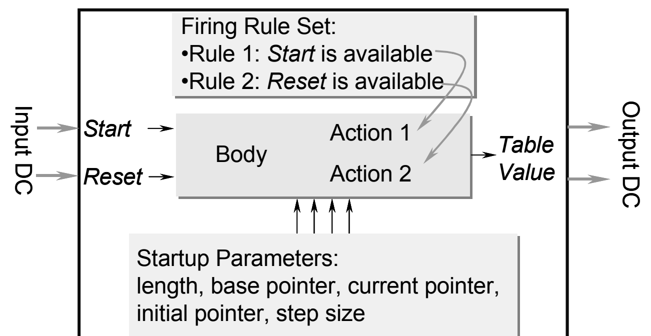

Fig. 6. Lookup_table ECO.

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.

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 6 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. Figure 7 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 Figure 7.

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

Fig. 7. Configuration block of Lookup_table.

The ECO's implementation, also called its body, is the program code that describes its behavior. Fig. 8 shows the implementation for the Lookup_table ECO.

```
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;                          (a)
        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 )            (b)
          config->Index = 0;
        write_float_DC( p->out_port[LOOKUP_TABLE_DATA], data );
        break;
    }
  }
}
```

Fig. 8. ECO body of Lookup_table.

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.

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 box (a) of Figure 8. If *Reset* data is available, it will cause another action to be executed. In this action, shown in box (b) of Figure 8, 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. Figure 8 shows the body of Lookup_table ECO. If input "Start" is available, action (a) gets executed; if input "Reset" is available, action (b) 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.

B. Data Channel Design

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. Figure 8 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.

Figure 9 shows the content of the data channel descriptor, which defines the channel's properties. In Figure 9, "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.

```
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;
```

Fig. 9. Descriptor that defines a 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. And 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.

Tab. 1. Performance analysis for the 3-phase inverter closed loop control application.

		Main-program-and-subroutine	Dataflow architecture
Execution time for one switching period (instruction cycles)	Computation	463	1195
	Data channel operations	-	1176
	ECO scheduling	-	462
	Context switching	-	263
	Total time	463	3296
Memory (Words)		1.1K	14.3K

V. Experimental Results

Though the dataflow architecture approach shows the benefits of reusability and reconfigurability, the main concern of using dataflow architecture is performance overhead introduced by system maintenance and communication between independent software components. To assess the performance of the dataflow architectural software, the control software for the 3-phase inverter shown in Figure 2 are constructed in dataflow architecture and main-program-and-subroutine architecture, both in C programming language.

Table 1 presents the performance comparison data for the closed-loop current control application for the three-phase inverter. From Table 1, it is clear that interprocess 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.

VI. Conclusion

This paper illustrates how to implement dataflow software for power electronics systems using a PEBB-based 3-phase inverter as an application example. 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. The main disadvantage of the dataflow architecture is the performance overhead introduced into the system, shown in the experimental comparison with main-program-and-subroutine approach. Further software optimization need to be exploited to reduce the overhead introduced by dataflow approach.

References

[1] J. Guo, S.H. Edwards, and D. Borojevic, “Elementary Control Objects: Toward a Dataflow Architecture for Power Electronics Control Software”, *33rd Power Electronics Specialists Conference,* Queensland, Australia, 2002.

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

[3] 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.

[4] 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, 2001.

[5] 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.