DAVID B. STEWART

Software Components for Real Time

Create your own framework for component-based real-time software without a huge cost, effort, or run-time overhead.

Component-based software helps you get a system working quickly, keep costs down, and reuse the most robust software from prior applications. This article presents methods for creating your own framework for component-based real-time software without the huge cost, effort, or software overhead associated with using commercial tools that are dedicated to this task. Any C programming environment can be used to create components with minimal increase in CPU or memory usage. The discussion will focus on techniques for modular decomposition, detailed design, communication, synchronization, scheduling, I/O drivers, and real-time analysis. The solutions can be implemented as a layer above your favorite RTOS, or stand-alone for performance- and memory-constrained applications that do not use an RTOS. The techniques have been demonstrated on a variety of microcontrollers and general-purpose processors. They've been used in applications including robotics, locomotive control, amusement devices, consumer electronics, and satellite modems.

A component-based software paradigm can be used effectively in the design of embedded real-time systems to provide advantages such as software reuse, improved maintainability, reconfiguring software on the fly, and ability to easily fine-tune a real-time application's timing properties. A more detailed discussion of the advantages to using component- based software is given in.[15]

In this paper, we present techniques for developing the solid framework needed to support component-based software, using the port-based

Reconfigurable components are modular components with the highest degree of modularity. Most important, they are modules designed to have replacement independence.

object (PBO) abstraction of a component. The techniques do not require any special commercial CASE (computer-aided software engineering) tools, and are compatible with most integrated development environments and RTOSes. For low-end processors without an RTOS, the methods can also be implemented using the dynamic scheduling real-time executive that is described in this article.

Some people believe that software reuse in embedded systems is near impossible; nothing can be further from the truth. It does not take tremendous experience or knowledge. Rather, the most important quality needed by the software designer and programmer to create reusable component-based software is *discipline*. This article provides the details on how to create such a software system; the discipline is required to follow some of the rules. The rules do not limit what can be done—they do limit how it is done—to ensure that the software can be reused. To enforce the discipline, formal design and code inspections should be performed at each step during the design and implementation phases.[1] Time spent on these reviews can easily save five to 10 times as much time debugging, both before and after deployment.

Background

Modular vs. reconfigurable software

Modular software is characterized by many guidelines, which include a simple structure, data encapsulation, functional and informational cohesion, separation of the interface specification, and the internal behavior implementation.[10,11,14] The *degree of modularity* refers to a subjective measurement that describes the extent to which a software module follows these guidelines. For example, a system decomposed into modules may be classified as "somewhat modular" or "highly modular," depending on a software engineer's assessment of how well the module meets the defined criteria.[3]

Reconfigurable components are modular components with the highest degree of modularity. Most important, they are modules designed to have replacement independence. In a modular system, there is often only one way to piece all the components together, because the interfaces of modules that need to be integrated are designed according to the other modules they interact with. For example, if a C or C++ module is written and a `.h` file of another module is `#include`d, then the module becomes dependent on the interfaces of that other module. In contrast, interface specifications for reconfigurable components are designed according to a pre-defined standard, not according to the interfaces of other modules with which it will be integrated. Interaction between components occur through these standard interfaces only.

By the above definitions, software components designed according to the PBO model are reconfigurable, because they are modular and have replacement independence.

Generic vs. reconfigurable software

Reconfigurable software does not necessarily imply generic software, for which it is sometimes mistaken. It is possible to have both hardware- and application-dependent components that are not generic, but are reconfigurable. Classifications of reconfigurable software components are defined in this section.

A *generic component* is a module that is neither hardware dependent nor application dependent. The component can be configured for different types of hardware, and can be used in different applications.

Hardware-dependent (HD) components are software modules that can only be executed when specific hardware is part of the system. HD components can be of two types: interface components and computation components.

HD interface components are used to convert hardware-dependent signals into hardware-independent data, such that other components can interface with these modules. The HD interface components replace standard I/O device drivers, and provide an interface to application hardware such as robotic actuators, switches, sensors, and displays. They differ from RTOS I/O device drivers because as processes with their own thread of control, they have the same standard interface as other software components, rather than being defined as system calls that are called through the operating system. The difference between our device driver model and the traditional module is illustrated in Figure 1. An extensive study of this driver model is given in an article by M. Moy and myself, as found in the Real-Time Symposium Proceedings.[9]

HD computation components provide similar functionality as generic components, but with better performance or added functionality, due to hardware-specific optimizations or modifications of the generic component. Unlike the interface components, they do not communicate directly to hardware; they are simply dependent on having specific hardware as part of the system. Rather, they interface to the HD interface components through their input and output ports.

Application-dependent components are modules used to implement the specific details of an application. As the name implies, these components are not reusable across different applications. Ideally, these components are eliminated, since they must be redeveloped for each new application. Modules initially defined as application-dpendent components, however, can often be transformed into generic components if an algorithmic abstraction of the module's functionality is possible; this would result in hard-coded information being converted to variable data. The configuration data can then be obtained from the user through a tele-operating device or keyboard, from a previously stored configuration in non-volatile RAM or EPROM, from a file, or from an external subsystem, depending on the capabilities offered by the target hardware.

Port-based object model

The model of a software component described in this article is targeted specifically to embedded real-time control systems. However, the techniques to create a framework and reusable objects that plug into the framework can be applied to other applications as well.

The model is based on domain-specific elemental units to maximize usability, flexibility, and real-time predictability. A framework is designed that uses these elemental units as building blocks to incrementally create larger, more complex applications.

There are two distinct aspects to integrating components. One is to integrate the data paths from an architectural perspective, as described in this section. The other is to integrate the code through use of a framework process and objects that "plug in" to the framework.

The *independent process* is the elemental process model that underlies the software component.[1] An independent process does not have to communicate or synchronize with any other component in the system, making integration simple. A system that is composed only of independent components, however, is very limiting, because no means exist to share data or resources. Nevertheless, this extreme emphasizes a desire to keep the pieces of the application as self-contained as possible, by minimizing the dependencies between components. The less dependencies a component has, the simpler it will be to integrate it into the system.

Streenstrup, Arbib, and Manes formalized the algebra of independent concurrent processes with their port-automaton theory.[13] They model a concurrent process as an independent automaton that operates on the state of the environment. The communica-

FIGURE 1 Comparison of traditional and new device driver designs. The new model leverages the fact that real-time software can be implemented as a multitasking application, thus the driver itself can have its own thread of control. The driver uses a data-driven approach in (b) to interact with the rest of the application software, as opposed to the more traditional process-driven approach shown in (a)

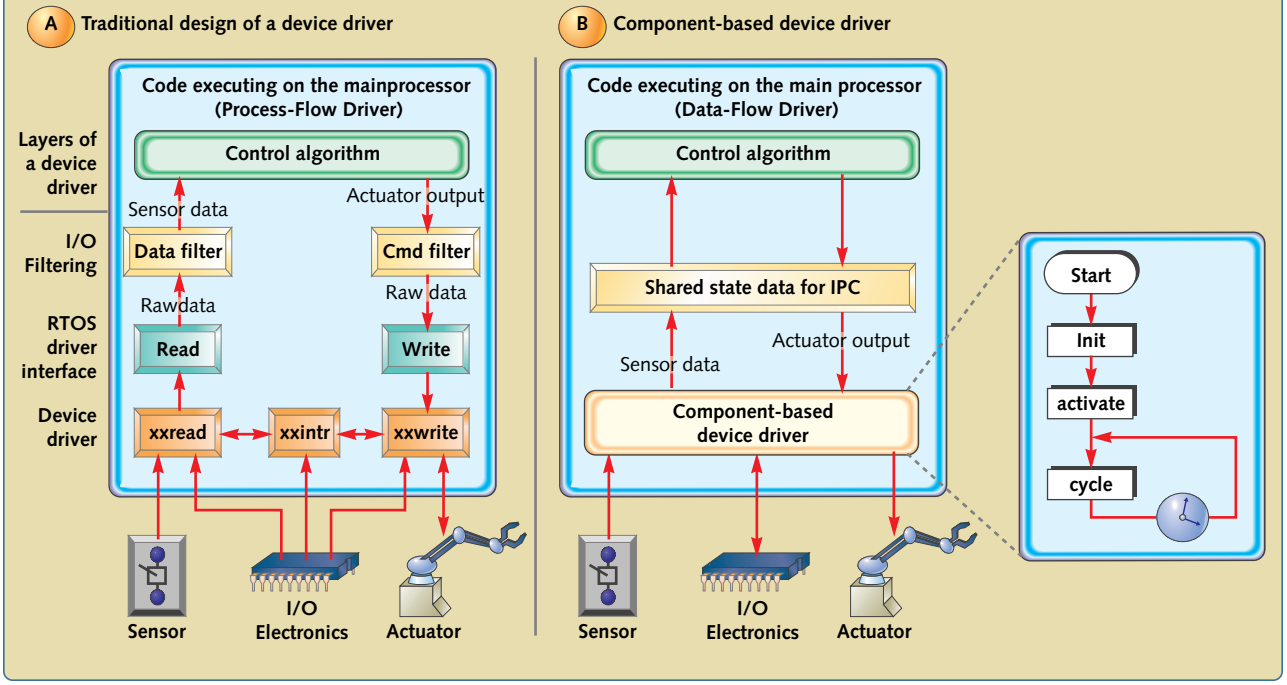


FIGURE 2 Diagrammatic model of a port-based object

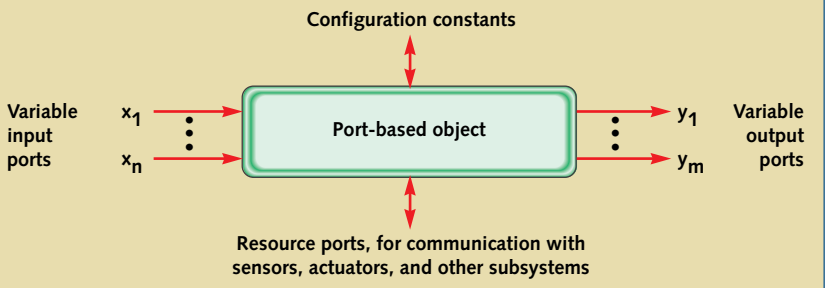


TABLE 1 Example of PBO software components in a library

Name	Function	Description
rmms	Robot interface	Hardware-dependent interface to the Reconfigurable Modular Manipulator System (RMMS)
gfwdkin	Generalized forward kinematics	Compute Forward Kinematics based on the DH-parameters obtained during initialization of the module
ginvkin	Generalized inverse kinematics	Computer Inverse Kinematics based on the DH-parameters obtained during initialization of the module
tball	Trackball interface	A hardware-dependent interface to a six-degree of-freedom trackball
cinterp	Cartesian trajectory interpolator	Given the current measured position and the desired final position, compute the intermediate reference positions
puma	Puma interface	Hardware-dependent interface to Puma 560 robot
pfwdkin	Puma forward kinematics	Compute forward kinematics for a Puma 560 robot
pinvkin	Puma inverse kinematics	Compute inverse kinematics for a Puma 560 robot

tion between objects is based on a structured blackboard design that operates as follows.

When a process needs information, it obtains the most recent data available from its *input ports*. This port can be viewed metaphorically as a window in your house; whatever you see out the window is what you get. There is no synchronization with other processes and there is no knowledge as to the origin of the information that is obtained from this port.

When a process generates new information that might be needed by other processes, it sends this information to its *output ports*. An output port is like a door in your home; you can open it, place items outside for others to see, then close it again. As with the input ports, there is no synchronization with other processes, nor do you know who might look at the information placed on the output ports.

In addition to the independent process, the *object* is selected as an elemental software abstraction. As stated by Wegner, an object is the atomic unit of encapsulation, with operations that control access to the data.[21] The term object does not imply "object-oriented design," which is an extension to objects to include *polymorphism* and *inheritance*. The references to objects in this article are classified as *object-based design*, as defined by Wegner's distinction of that term and *object-oriented design*.[22] Note that objects without inheritance and polymorphism are in effect abstract data types (ADTs), and are easily implemented in C; C++ is not necessary.

The algebraic model of a port automaton and the software abstraction of an object are combined to create the PBO model, as depicted in Figure 2. A PBO is drawn within a data-flow diagram as a round-corner rectangle, with input and output ports shown as arrows entering and leaving the side of the rectangle. Configuration constants are drawn as arrows entering/leaving the top of the rectangle. Resource ports are shown as arrows entering/leaving the PBO from the bottom.

A PBO executes as an independent concurrent process, whose functionality is defined by methods of a standardized object. In C, the objects are implemented as ADTs. Communication with other modules is restricted to its input ports and output ports, as described above. The *configuration constants* are used to reconfigure generic components for use with specific hardware or applications.

In addition to input and output ports, we also define *resource ports*, which are needed to create an environment for multi-sensor integration. The resource ports are for modeling only to show the source or destination of data that is exchanged with I/O hardware. In practice, the resource ports are implemented in a hardware-dependent manner, as the reads and writes of the I/O hardware's registers. The resource ports connect to sensors and actuators, allowing the PBO model to be used to replace the more traditional POSIX style of device drivers. Details of accessing the sensor or actuator are encapsulated within the PBO, resulting in an HD interface component.

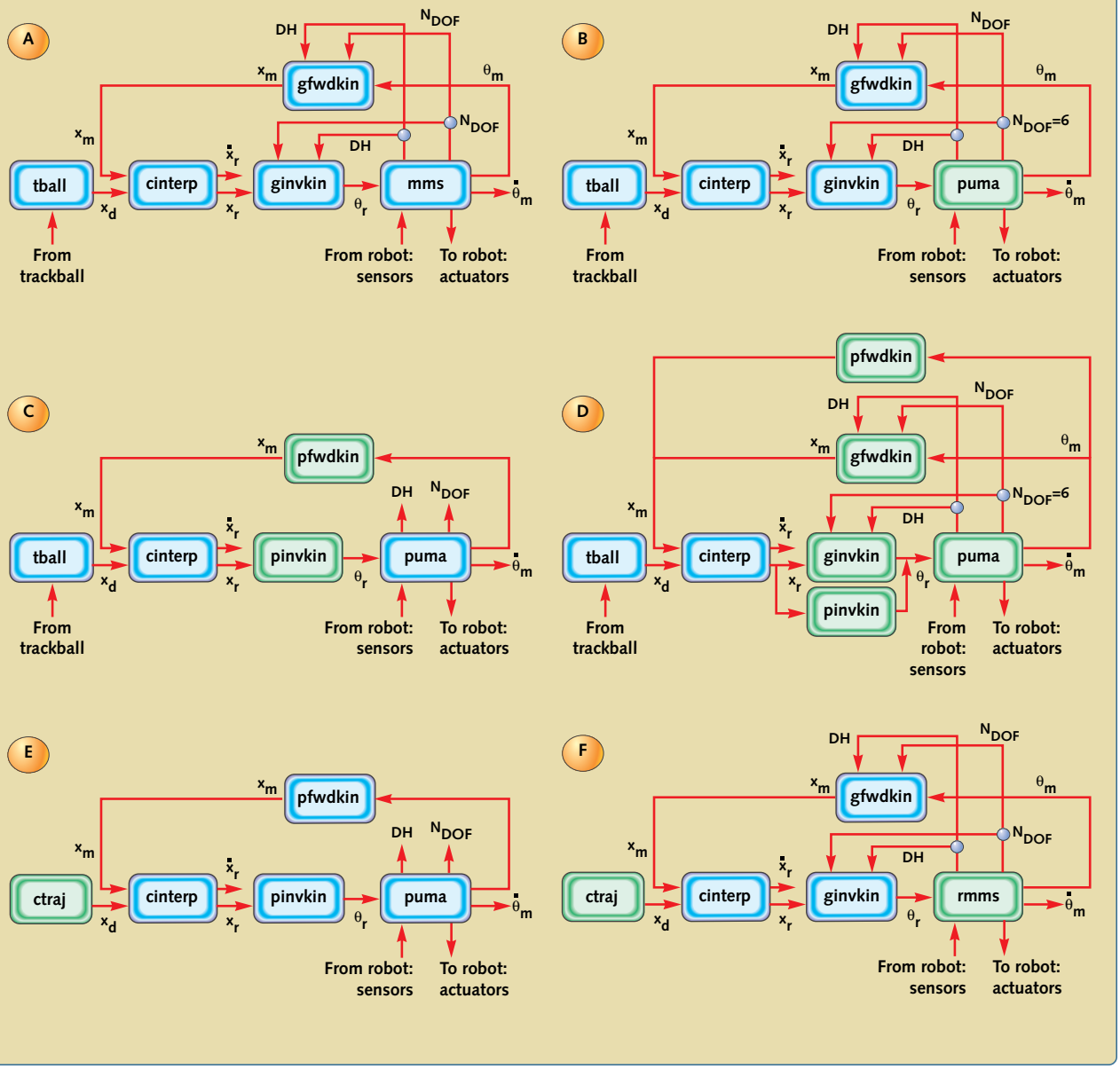
Modelling PBOs to have optional configuration constants and resource ports allows the use of the same PBO model for different types of components. A sample library of PBO objects for robotic manipulators is shown in Table 1. The library represents a subset of PBOs that were created in a robotics laboratory at Carnegie Mellon University.[19]

An important note about the functional descriptions of the modules is that the framework is designed independent of the granularity of functionality in each PBO. The granularity is defined by the software

architect who decomposes an application into modules; the framework then provides the mechanisms for quickly realizing each of these modules by using the PBO model to implement them as reconfigurable objects.

Similarly, the framework does not define the type nor semantics of the port variables. A variable type mechanism is used so that data transmitted over the ports can be any type. For example, it can be raw data, such as input from an A/D converter; processed data, such as positions and velocities; or processed information, such as structures describing types and locations of objects in the environment. The names of the ports are configurable, and specified during the initialization of the system.

FIGURE 3 Example of component-based design using port-based objects. (a) Cartesian teleoperation of the RMMS using generic components. (b) Cartesian teleoperation of a Puma 560 using generic components. (c) Cartesian teleoperation of a Puma 560 using HD computation components. (d) Example of fault-tolerant components, with coexistence of old and new software components. (e) Example of application-specific autonomous execution of the RMMS using generic components. (f) Example of application-specific autonomous execution of a Puma 560 using HD computation components



A configuration is a set of PBOs that are interconnected to provide the required open-loop or closed-loop system.

FIGURE 4 Comparison of traditional process model and port-based object process model

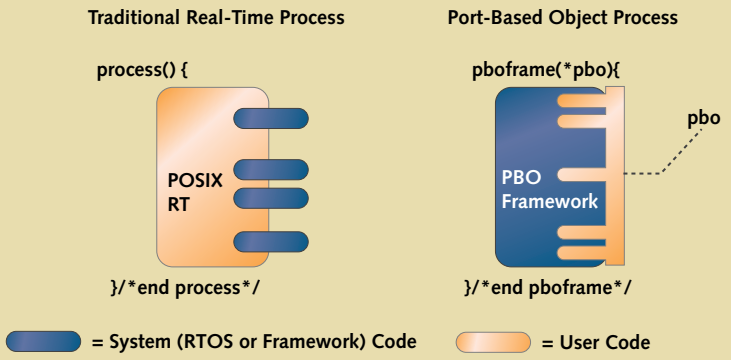
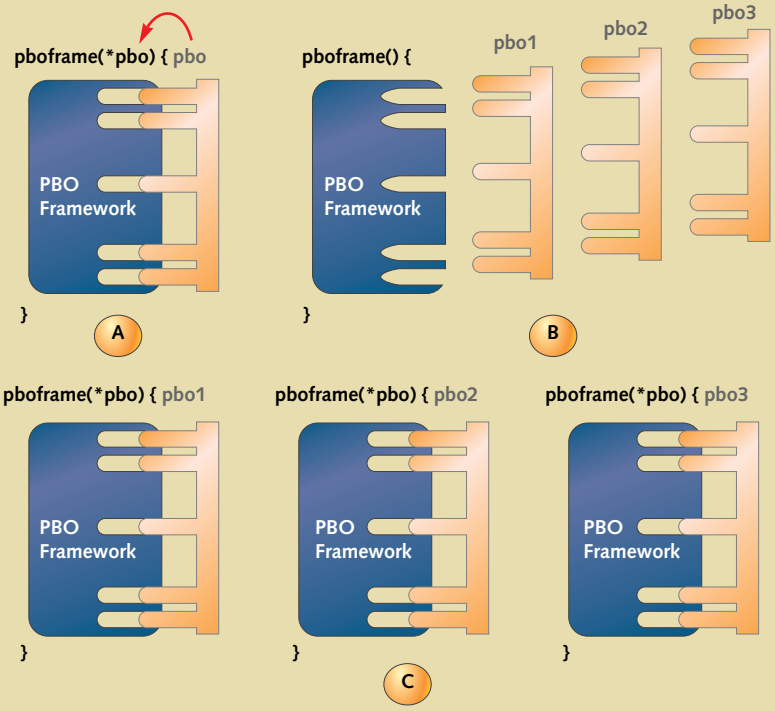


FIGURE 5 (a) A PBO "plugs in" to the framework. (b) In a non-preemptive environment, there is one context for the framework process; each PBO plugs into that context when it is its turn to execute. (c) For the preemptive case, the context of the framework process is replicated and initialized to execute one of the PBOs in the system



Configurations

As defined by Dorf, "a control system is an interconnection of components forming a system configuration which will provide a desired system response."[4] Each component can be mathematically modeled using a transfer function to compute an output response for any given input response. The port-automaton theory provides an algebraic model for these types of control systems. By incorporating the model into the PBO, the PBOs provide a model suitable for control engineers. PBOs are configured to form a control system in the same way that a control engineer configures a system using transfer functions and block diagrams. This approach allows the framework to satisfy an important criterion: to make it easy to program for a target audience of control engineers who do not have extensive training in software engineering or real-time systems programming.

A *configuration* is a set of PBOs that are interconnected to provide the required open-loop or closed-loop system. A configuration is valid only if for every PBO selected, any data that it requires at its input ports is produced by one of the other PBOs as output. As per the port-automaton theory, the control engineer does not have to be concerned with how data gets from the output of one PBO to the input of another PBO. The communication is embedded in the framework, such that it is transparent to the control engineer. A configuration also cannot have two PBOs that produce the same output, otherwise a conflict may arise as to which output should be used at a given time.

Port names are used to perform the bindings between input and output ports. Whenever two PBOs exist with matching input and output ports, the framework creates a communications link from the output to the input. If necessary, the output can be fanned into multiple inputs. Our framework uses an internal/external name separation for the ports, such

that the name used to code the PBO can be independent of the name used for linking that object to other PBOs.

Configuration examples

The flexibility of creating applications using software components is demonstrated in this section. Details of designing individual PBOs are given in a following section.

Cartesian control of the RMMS

Figure 3a shows a configuration, using modules from our sample library shown in Table 1, to perform teleoperated Cartesian control of a reconfigurable modular manipulator system (RMMS).[12] The configuration of this robot is not known beforehand. Rather, its configuration is read from EPROMs embedded in the robot during initialization. From that configuration, the *rmms* module outputs its hardware configuration via the N_{DOF} (number of degrees) and *DH* (Denavit-Hartenberg parameters—a method of mathematically specifying the shape of a robot) configuration constants. The constants are used as input to the *gfwdkin* and *ginvkin* modules that are configured for any robot based on N_{DOF} and *DH*.[6] A teleoperation interface is provided by the 6-DOF trackball, and the *cinterp* module is used to generate intermediate trajectory points for the robot, because the *tball* module typically executes at a much lower frequency than the other modules.

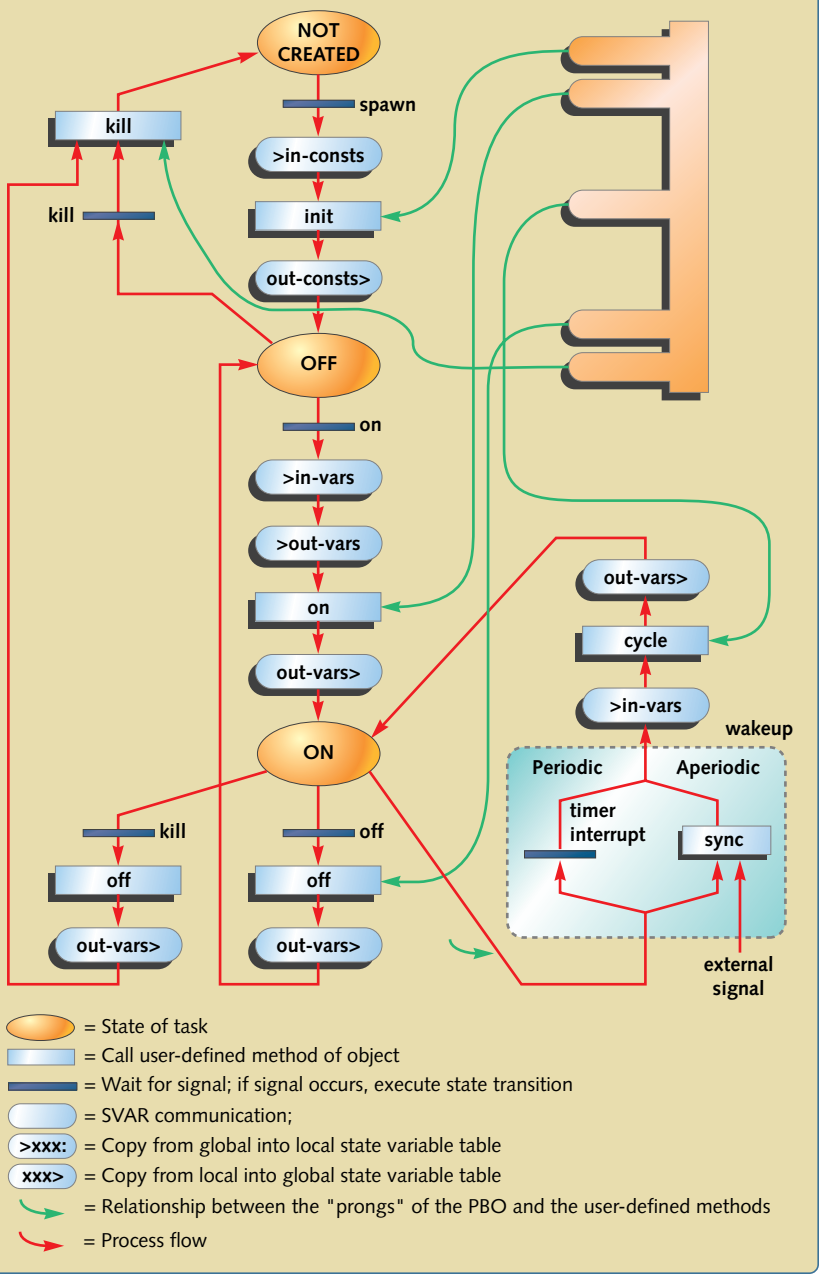
The software framework does not pose any constraints on the frequency of each PBO. Rather, as defined by the port-automaton theory, every PBO is an independent, concurrent process that can execute at any frequency. Whenever that process needs data from its input ports, it retrieves the most recent data available. When it completes its processing, it then places any new data onto its output ports.

A configuration can be executed in either a single- or multi-processor environment. In a multi-processor environment, the control engineer only needs to specify which processor to use for each PBO. The communication between PBOs and synchronization of their processes is otherwise identical, and fully transparent to the control system engineer.

Cartesian teleoperation of a Puma 560

Suppose that a Puma 560 robot is to be used instead of the RMMS. The *rmms* module can be replaced with the *puma* robot interface module, as shown in Figure 3b. Since the Puma is a fixed configuration robot, its N_{DOF}

FIGURE 6 Finite state machine and process flow diagram describing the internals of the framework process. Transitions between states are process flow diagrams. Each prong of the object that plugs into the framework is a user-defined function. The rest of the code is part of the framework



LISTING 1 Declarations for PBO framework. The *pboFunc_t* type and *PBO_MODULE* macro as shown here should be defined in *pbo.h*

```
typedef int (* pboFunc_f)(void *);

typedef struct _pboFunc_t {
  pboFunc_f init;
  pboFunc_f reinit;
  pboFunc_f on;
  pboFunc_f cycle;
  pboFunc_f sync;
  pboFunc_f off;
  pboFunc_f term;
}

#define PBO_MODULE(xyz)\
  typedef int (* xyz##Func_f)\
    (xyz##_t *);\ \
  int xyz##Init(xyz##_t *);\
  int xyz##Reinit(xyz##_t *);\
  int xyz##On(xyz##_t *);\
  int xyz##Cycle(xyz##_t *);\
  int xyz##Sync(xyz##_t *);\
  int xyz##Off(xyz##_t *);\
  int xyz##Term(xyz##_t *);\
\
typedef struct\  ##xyz##Func_t {\
  xyz##Func_f    init;\
  xyz##Func_f    reinit;\
  xyz##Func_f    on;\
  xyz##Func_f    cycle;\
  xyz##Func_f    sync;\
  xyz##Func_f    sync;\
  xyz##Func_f    off;\
  xyz##Func_f    term;\
} xyz##Func_t;\
\
const xyz##Func_t\
  xyz##Func={\
  xyz##Init,\
  xyz##Reinit,\
  xyz##On,\
  xyz##Cycle,\
  xyz##Sync,\
  xyz##Off,\
  xyz##Term\
};
// end PBO_MODULE definition
```

and *DH* parameters are constant. Instead of reading these values from the robot, they can instead be hard-coded into the *puma* module, and output as configuration constants. There is no need to change any other module, since the *gfwdkin* and *ginvkin* modules will configure themselves during initialization for the Puma based on the new values of N_{DOF} and *DH*.

Improving performance of a Puma 560

Generic components are useful for enabling rapid prototyping, but they may not always be computationally efficient. For example, the generalized computation of the forward kinematics (module *gfwdkin*) is based on the DH configuration constants and using matrix operations. This will naturally be slower than performing similar computations for a specific robot, such as the Puma 560, where the DH parameters are constant. Unnecessary computations (such as multiply by zero or one, or computing $sin(\pi/2)$) can be eliminated.

An HD computation component can be created to improve the performance of an application. The *pfwdkin* and *pinvkin* modules are examples of such components. They compute the forward and inverse kinematics specifically for a Puma 560, and they execute faster than their generic counterparts. It is then desirable to replace *gfwdkin* with *pfwdkin*, and *ginvkin* with *pinvkin*, as shown in Figure 3c, whenever the *puma* HD interface component is used.

In order for an HD computation component to replace a generic component, it must provide at least the same outputs and must not require any additional inputs as compared to the generic component. Even when an HD component is used, it does not eliminate the usefulness of the generic component. For example, in order to improve fault tolerance of an application, the generic component can still be used as a standby module, or as shown in Figure 3d, it can execute in parallel with the HD computation component, albeit at a lower frequency, in order to provide consistency checks.

Autonomous execution of a Puma 560

As an example of an application component, suppose that a custom autonomous trajectory module *ctraj* is created to replace the teleoperation module *tball*, as shown in Figure 3e. The component can be integrated into the system by defining it as a PBO.

Even though a module is application dependent, it does not have to be hardware dependent. If the hardware for the application is changed, the application component does not necessarily have to change. Figure 3f shows this by replacing *puma* with *rmms*, but not changing the trajectory of the robot's end effector, as defined by *ctraj*.

Overview of framework

Creating code using the PBO methodology is an "inside-out" programming paradigm as compared to traditional coding of real-time processes, as shown in Figure 4.

The traditional approach is used by most current RTOSes. Processes are created, each with their own `main()` (or equivalent function name). The process executes user code and controls the flow of the program. It invokes the RTOS, typically via a system call, whenever an RTOS service is required. RTOS services include communication, synchronization, programming timers, performing I/O, and creating new processes. Because execution is under the control of the user, it forces the user to be responsible for all of the resource management, such as scheduling, communication, and synchronization.

Instead, to achieve the consistency needed to support component-based software, the RTOS must serve as the resource manager. The RTOS has control of execution at all times, and performs the communication, synchronization, scheduling, and process management in a predictable manner.

Component-based software support is realized by creating a single, standard process that we call the framework process (pboframe).

Only when necessary, the RTOS invokes a method of one of the software components to execute application code.

In this section, we provide more details on the framework. In particular, we show how to create a framework for both periodic and aperiodic tasks, in both non-preemptive and preemptive environments. Typically a non-preemptive approach is used for low-end processor environments that cannot afford the overhead of a full RTOS. The framework for the preemptive approach can be implemented as middleware, to operate with your favorite RTOS. The difference in the approaches is illustrated in Figure 5. The non-preemptive case has only a single context, and each object is plugged in as it is needed. For the preemptive case, the context of the framework is replicated for each task, and a PBO is bound to that framework for the lifetime of the task. The details of the framework follow in the remainder of this section.

The framework process

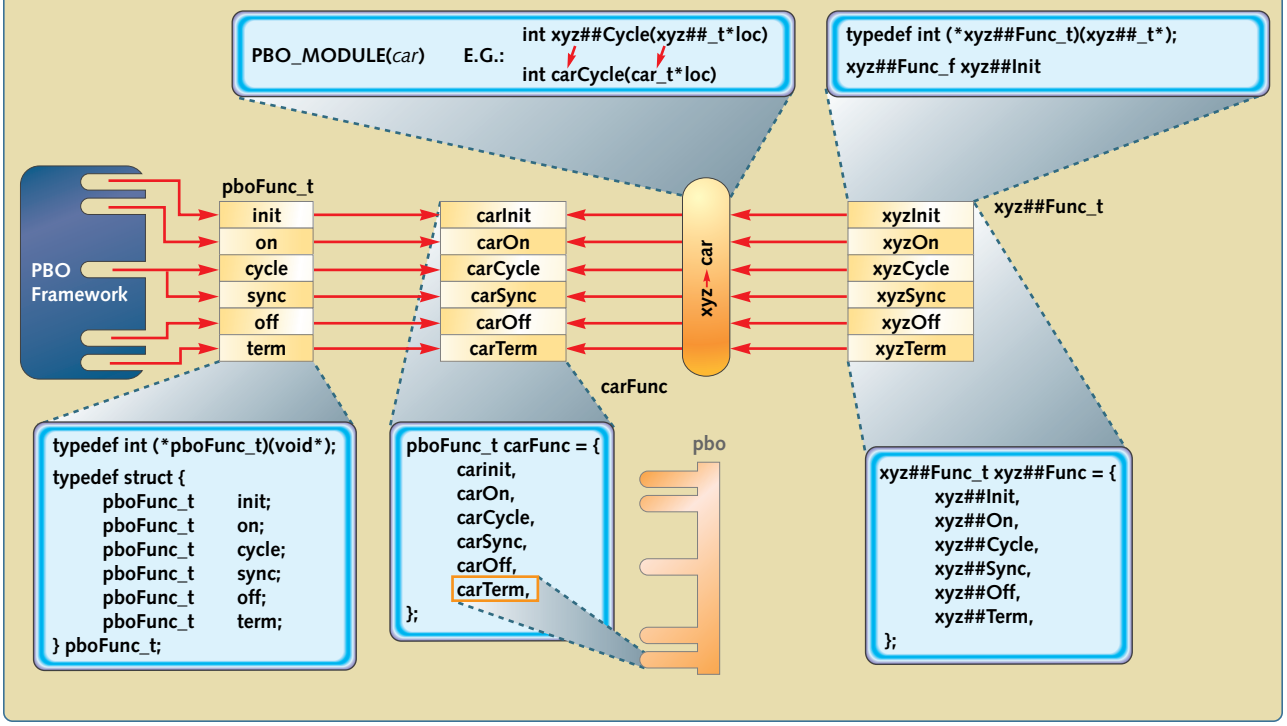
Component-based software support is realized by creating a single, standard process that we call the *framework* process (*pboframe*). Both periodic and aperiodic processes in the system use this same framework. The process *pboframe* takes a PBO as an argument. The PBO defines the module-specific code, including the input and output ports, configuration constants, the type of process (for example, *periodic process* or *aperiodic server*), and the timing parameters such as frequency, deadline, and priority.

The framework process implements a finite state machine with three states, as shown in Figure 6. The states are shown as bold ellipses, and are `NOT_CREATED`, `ON`, and `OFF`. Extensions to include an error state can be found in [19]. State transitions are shown in the diagram as process flow diagrams. A state transition is triggered by a signal (drawn as solid bars). Signals may originate from interrupts, a planning module, an external subsystem, or from the user through a graphical user interface.

In response to a signal, a data transfer is made to receive data from other objects. One of the user-defined functions is then called, followed by another transfer to send data to other objects.

The PBO method that is called depends on the state of the process and the signal that is received. For example, if a PBO is in the `ON` state, and it receives a *wakeup* signal, then it will execute the *cycle* method and remain in the `ON` state. On the other hand, if the PBO is in the `ON` state, and receives the *kill* signal, then it will execute the *off* method, followed by the

FIGURE 7 Expansion of the PBO_MODULE macro used to enforce the PBO API and bind the PBO to the framework



kill method, then enter the `NOT_CREATED` state.

The framework process, as shown, evolved over several years as we designed and tested many variations, in order to obtain a common program structure for all software components. The diagram represents the most recent revision in the evolution of the structure. Many design decisions are implied by the detailed PBO framework, as now discussed.

Despite the seeming complexity of the framework, dissecting it into pieces shows that it is indeed rather simple. In the steady state, PBO processes are all in the `ON` state, executing their *cycle* method once per cycle or event, and going back to sleep until the next *wakeup* signal. Note that the only difference between a periodic process and an aperiodic server is the source of the wakeup signal. For a periodic process, the wakeup signal is received from the clock, as a virtual timer interrupt. For the aperiodic processes, the process blocks on a semaphore, message, or event, as defined by the *sync* method of the PBO.

The autonomous nature of the PBO allows the most popular scheduling algorithms, such as the rate monotonic static priority,[7] earliest-deadline-first,[7] or maximum-urgency-first dynamic priority scheduling[20] algorithms, to be used to schedule PBOs. The user can also choose between a preemptive or non-preemptive environment, and still use the PBO model. The control systems designer only needs to specify the frequency of the *cycle* routine for each PBO based on the needs of the application.

By using the timing error detection and handling methods described in an article called "Mechanisms for Detecting and Handling Timing Errors" by myself and P. K. Khosla, [16] aperiodic servers can use the same fundamental structure as periodic processes. The framework can define aperiodic processes as either deferrable or sporadic servers, and use them with either the rate monotonic static priority or maximum-urgency-first dynamic priority scheduling algorithms to ensure predictable scheduling.

The remainder of the framework handles the initialization, termination, and reconfiguration. To support dynamic reconfiguration, a two-stage initialization and termination is used. High-overhead initialization of a new process can be performed upon system start-up in preparation for being activated. The initialization includes creating a process's context, dynamically allocating its memory, binding

LISTING 2 Example of the API for the tball component, showing header and function definition of the init() and cycle() routines. Most important to note is that the PBO_MODULE(tball) expands to all of the necessary function prototype definitions. This keeps the user module quite simple, and enforces that the PBO is defined with all the necessary methods and the proper arguments. The code in bold can easily be generated from a template; the user only needs to "fill in the blank" with code where the comments indicate

```
#include <pbo.h>

typedef struct {
  // Internal state for the module goes here.
} tballLocal_t;

PBO_MODULE(tball)

int tballInit(tballLocal_t *local) {
  // Module-specific initialization goes here
  return (int) local;
}

int tballCycle(tballLocal_t *local) {
  SVAR_OUT(xd);
  // Module-specific 'cycle' code goes here
  return PBO_OK;
}
```

input and output ports, and calling the user-defined *init* method. The process then waits in the `OFF` state, and can be viewed as being in a standby mode for a dynamic reconfiguration. When an `ON` signal is received, the local table is updated to reflect the current state of the system, and execution begins.

Implementation of the framework

The framework is defined in two parts: header information including data structures and function prototypes that enforce the interface between the PBOs and the framework, and the code that executes the FSM that was shown in Figure 6.

We provide a sample of the framework header file in Listing 1; this code would go in the file `pbo.h`, to be `#include`d by every software component. Note that this framework code only needs to be written once, and can be used from one application to another.

The header file defines the `pboFunc_t` structure that will contain a pointer to each of the functions of the PBO. `PBO_MODULE(xyz)` is a macro that expands into all of the declarations needed by the software component, and allows the compiler to enforce the API of the component. The expansion of this macro is one of the keys to creating component-based software, and is illustrated in more detail in Figure 7.

On the left side of this diagram, the framework process only knows about components based on their functions. In this example, the object "car" is plugged into the framework. To do so, the structure `carFunc` needs to be defined. This structure is defined by placing `PBO_MODULE(car)` in the declarations part of the file `car.c`. The right side of the diagram shows the generic form of the `PBO_MODULE` macro, and how substitution of `xyz` for `car` results in the desired declarations (`##` is the C preprocessor token pasting operator; it is part of ANSI C). The macro also defines the function prototypes for each method, so that when the code is compiled, if the user's code does not match what is expected by the framework, compiler warnings or errors will be generated.

A software component then only needs to `#include <pbo.h>`, and call the macro before defining the functions. An example is given in Listing 2. It is an excerpt of the module `tball`, that shows the header information, and the `init` and `cycle` functions (the two most complex functions of most modules).

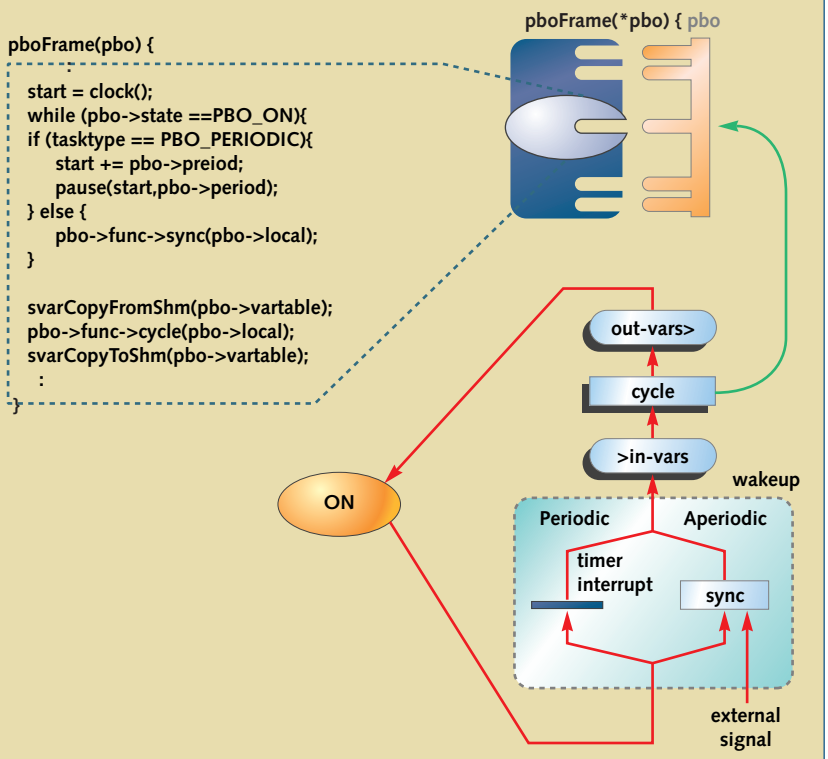
A list of the `xyzFunc` structure for each PBO can then be maintained, either statically (that is, an array) or dynamically (that is, a linked list). The list can also include additional information about the object needed for real-time scheduling, such as period and priority. In the non-preemptive case, the framework itself does scheduling and selects one of the objects from the list. In the preemptive case, a new thread is created for each object in the list, and the underlying RTOS takes care of all the scheduling.

Due to space limitations, it is not possible to show the entire code for the framework process shown in Figure 6. Instead, we show the framework code that surrounds calling the *cycle* routine, which is the main body of any object.

In Figure 8, the code for a preemptive system is shown. The `svarCopyXxShm` functions are the interprocess communication mechanism (more on this later). The code is a loop that blocks at the beginning. If the task is periodic, it calls `pause()`, which is an RTOS function to wait for the next start time.[16] If the task is aperiodic, it calls the `sync()` method, where user-defined blocking (such as waiting for a message) can be defined. When the signal is received, data is

Integrating software components such that all communication is performed in a predictable and timely manner is perhaps the most difficult aspect of creating component-based real-time systems.

FIGURE 8 Framework code surrounding the call to the PBO's cycle() routine for the *preemptive version*. Execute each PBO as a thread by binding each thread to the function pboFrame(), and passing the PBO as an argument. Note: this is pseudocode; some functionality is not listed, such as function return values and checking for change of state



copied from shared memory, processed by calling the object's `cycle()` routine, and the output data copied back into shared memory.

Figure 9 shows the framework for a non-preemptive system. While this code is a bit more complicated than for the preemptive case, keep in mind the code does the scheduling and eliminates the need for an underlying RTOS. The do-loop at the beginning of the main loop performs an earliest-deadline-first (EDF) scheduling to obtain the next task to execute. Other scheduling algorithms for non-preemptive systems can also be implemented at this point instead of EDF. Once the object to execute is selected, the `cycle()` routine is called. Since there is no preemption, there is no need for a complex IPC mechanism that maintains integrity of global data; instead, each object directly uses pointers to the shared data.

Summary of framework process

While the framework shown is designed for control system components. A similar approach can be used to integrate other types of components too. For example, a display subsystem would allow the user to plug in different screens or menus to quickly create and modify code for an embedded display device. The display manager itself is a PBO, which enables it to interface to the rest of the system. The cycle routine of the display manager can itself be a framework for a different class of objects, in this case menu objects.

The framework is specifically designed using C, because many embedded system programmers are not experts in object-oriented design or C++. Rather, they are experts in the application area, and are more familiar with C than C++. The framework likely uses less overhead than C++ objects, but we have not yet done actual performance comparisons. We have done performance benchmarking of the non-preemptive framework on an 8MHz Motorola 68HC12, and found that the overhead of switching between objects is less than 100µs.

Inter-object communication

Integrating software components such that all communication is performed in a predictable and timely manner is perhaps the most difficult aspect of creating component-based real-time systems. In order to support the PBO model, a communication mechanism was designed that meets the following requirements for embedded control systems:

- Support the port-automaton model of independent processes. That is, read the input ports at the beginning of each cycle to obtain the most recent data available, and write to the output ports at the end of each cycle
- Target data transfers with low volume (less than 100 bytes per transfer) but at high frequency (1,000Hz)
- Have a simple and straightforward binding scheme for input and output ports, to enable dynamic reconfiguration in bounded time
- Fan an output into multiple inputs

Every I/O port and configuration constant is defined as a state variable (SVAR) in the global table, which is stored in shared memory.

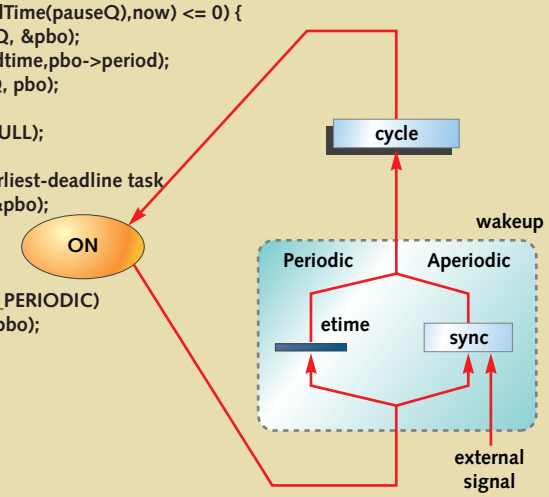
FIGURE 9 Framework code surrounding the call to the PBO's cycle() routine for the *non-preemptive version*. Execute a single instance of the framework. On each iteration, select the next PBO to execute, then call the cycle() routine of that PBO. Note: this is pseudocode. Some auxiliary functionality is not listed, such as function return values and checking for change of state. The etime module is an abstract data type for reading and manipulating timer data, while the EnqueueNode and DequeueNode are local functions for linked list manipulation

```
while (1) {
:
   do {
    etimeClock(&now);
      // Move newly awoken tasks from PauseQ to ReadyQ
    while (!Empty(pause!) &&
         etimeCompare(SchedTime(pauseQ),now) <= 0) {
      DequeueNode(&pauseQ, &pbo);
      etimeAdd(&pbo->schedtime,pbo->period);
      EnqueueNode(&readyQ, pbo);
    }
   } while (readyQ.next == NULL);

    //Head of readyQ has earliest-deadline task
   DequeueNode(&readyQ, &pbo);

   pbo->func->cycle(pbo);

   if (pbo->tasktype == PBO_PERIODIC)
     EnqueueNode(&pauseQ,pbo);
   else
     pbo->func->sync(pbo);
}
```



- Support transparent multiprocessing, so that objects can be either on the same or different processors, without any difference in the communication interface. Any differences must be encapsulated within the RTOS
- Allow communication between processes that may be executing at different frequencies

Communication between components may seem like a natural candidate for message passing. However, message passing was not used for several reasons:

- The port-automaton theory cannot be supported with messages because the most recent data is not always readily available. For example, if the process producing the data is faster, then the messages may be queued, and the message received by the consumer might not contain the most recent data
- Fanning an output to multiple inputs is difficult because it requires a message to be duplicated for each input or requires a more complex mechanism to ensure that no message is deleted until all processes needing it have used it. Duplicating messages based on the number of recipients also violates the port-automaton theory, which states that a process is unaware of the destination of the data on its output ports
- The overhead with sending messages—especially in a multiprocessor environment—is much higher than that achievable using shared memory. This factor is especially important considering some data must be transferred 1,000 times per second

These drawbacks of message passing systems led to the design of a mechanism based on shared memory, using the state variable table communication that is now described.

State variable communication

The communication between PBOs is performed via state variables stored in global and local tables, as shown in Figure 10. Every I/O port and configuration constant is defined as a state variable (SVAR) in the global table, which is stored in shared memory. Figure 10 shows the contents of the global and local tables for the sample configuration that was illustrated in Figure 3a.

A PBO can only access the local table, where only the subset of data from the global table that is needed by that PBO is kept. Since every PBO has its own local table, no synchronization is needed to read from or write to it. A PBO process can thus execute independently of other processes by using the data in its local table. Consistency between the global and local tables is maintained by the SVAR mechanism.

When a non-preemptive real-time executive is used instead of an RTOS, there is no need for the local table, as shown in Figure 11. Each PBO points directly to the global table, and proper mutual exclusion is achieved through not allowing preemption in the middle of a process's *cycle* routine. This significantly reduces overhead, allowing for the PBO model to be used

on low-performance microcontrollers with very limited memory. Since no copying occurs between local and global tables, the remainder of the discussion in this section focuses on the mechanism suitable for preemptive systems; the mechanism is not required for the non-preemptive case.

Support for SVAR communication is built into the framework, such that updates of the local and global tables occur at predetermined times only, as was shown by the ovals in Figure 6. Configuration constants are updated only during initialization of the PBO. The state variables corresponding to input ports (called `INVARS`) are updated prior to executing each cycle of a periodic PBO, or before processing each event for an aperiodic PBO. During its cycle, a PBO may update the state variables corresponding to output ports (called `OUTVARS`) at any time. These values are only updated in the global table after the PBO completes its processing for that cycle or event. All transfers between the local and global tables are block transfers (that is, using a routine like UNIX's `memcpy()`). Ensuring the integrity of the data is a matter of ensuring that the block transfers are performed as critical sections.

One of the keys to supporting dynamic reconfiguration of software components is correctly initializing the port variables. As shown in Figure 6, before calling the *on* method, it is necessary to read the `OUTVARS` in addition to the `INVARS`. This ensures that when a process is activated, its view of the environment is correct. The need arises because if the process is not yet executing, it is possible that some other process is generating those `OUTVARS`. In general, a process that outputs `OUTVARS` knows the same values on the subsequent cycle because they are in the local table. However, in a process's first cycle, this is not the case. Ultimately, rectifying the situation is a simple matter of also reading the `OUTVARS` during activation, thus updating the local table to properly reflect the state of the system. The *on* method of the process is then called, such that if necessary, these `OUTVARS` can be updated, before the process enters the on state.

Although there is no explicit (that is, apparent to the user) synchronization or communication among processes, it is still necessary to ensure that accesses to the same SVAR in the global table are mutually exclusive. The locking mechanism allows the autonomous execution model of the PBO to be maintained, while ensuring the integrity of the communication.

FIGURE 10 Structure of SVAR mechanism for PBO integration for a multiprocessor preemptive system

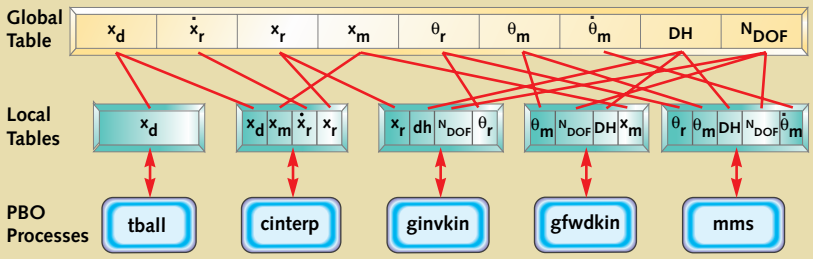
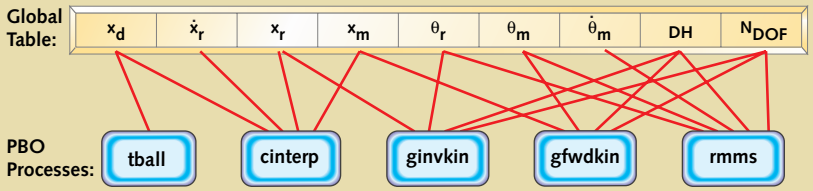


FIGURE 11 Structure of SVAR mechanism for port-based object integration for a single processor with no preemption



Locking the global SVAR table

The method used to lock the global table and preserve the autonomous execution model of the PBO is based on the assumption that the amount of data communicated via the ports on each cycle of a PBO is relatively small. That is, each `INVAR` or `OUTVAR` is only a few tens to a few hundreds of bytes. The exact value of what is meant by "small" depends on a particular configuration, and is quantified in an article that I and others wrote for the *IEEE Trans. on Software Engineering*.[19] As a rule of thumb, less than 100 bytes of communication between two objects per cycle is usually sufficiently "small." For this reason, the framework that is suitable for the control systems domain is not directly applicable to a general communication system with kilobytes or megabytes of data being transferred. The non-preemptive framework, however, is suitable even for a high volume of data exchange between objects, because there is no data replication.

The global table can be accessed by processes executing on different CPUs, making a solution for locking the table that is also suitable for multiprocessor environments desirable. The solution used is based on spinlocks.[8] See again my article from the *IEEE Trans. on Software Engineering* for a discussion of other solutions that were considered but not used—such as the shared memory protocol and priority ceiling protocol—because they are impractical for most embedded applications due to their high overhead.[19]

When a task must access the global table, it first locks the processor on which it is executing. Locking the CPU ensures that the task does not get swapped out while holding the critical global resource. The task then tries to obtain a global lock by performing an atomic *read-modify-write* instruction, which is supported by most processors. If the lock is obtained, the task reads or writes the global table then releases the lock, while remaining locked into the local CPU. It then releases its lock on the local processor. If the lock cannot be obtained because it is held by another task, then the task spins on the lock. It is guaranteed that the task holding the global lock is on a different processor, and will not be preempted, thus it will release the lock shortly.

In theory, locking the CPU can lead to possible missed deadlines or priority inversion. However, considering the practical aspects of real-time computers, it is not unusual that a real-time microkernel locks the CPU for up to 100µs in order to perform system calls such as handling timer interrupts, scheduling, and performing full context switches.[17] Furthermore, many RTOSes are created such that periods and deadlines of processes are rounded to the nearest multiple of the system clock since more accurate timing is not available to the scheduler. In these systems, if the total time that a CPU is locked to transfer a state variable is small as compared to the resolution of the system clock, then there is negligible effect on the predictability of the system due to this mechanism locking the local CPU.

When using the non-preemptive version, there is no need to lock the global SVAR table.

Decomposition of subsystem into components

To develop component-based software, it is necessary to draw clean boundaries between each component. Many attempts at creating component-based software fail because designers have no guidelines to follow when breaking up the application into pieces. In this section, general guidelines for decomposing the subsystem into components are presented. It is assumed that the application has already been split into different subsystems based on functionality.

A subsystem is defined as part of an application that can be developed and tested independently, and integrated into an application later through sim-

FIGURE 12 Example of an output element, drawn as a PBO software component

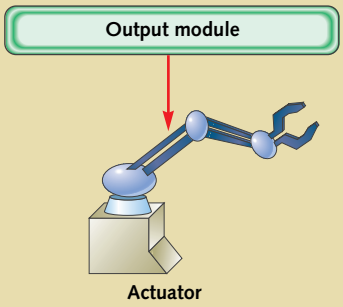
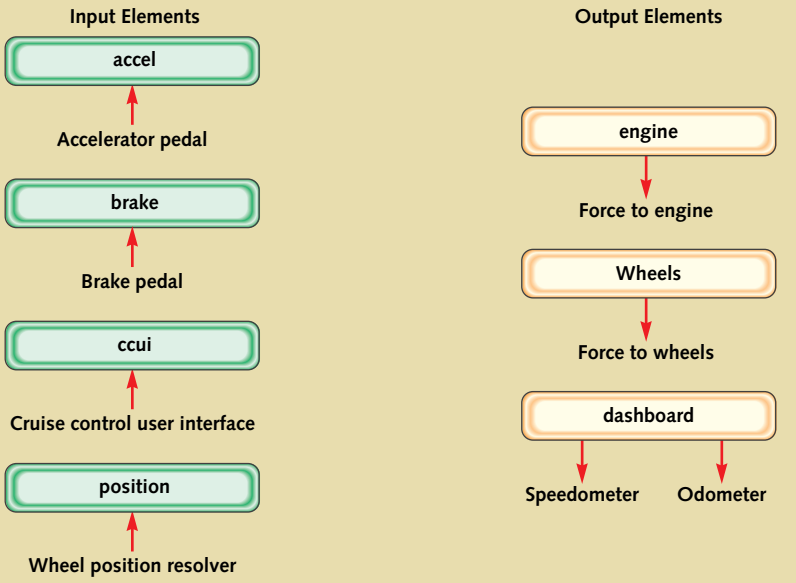


FIGURE 13 First step in decomposition: drawing of I/O elements



ple communication. For example, in a manufacturing plant, the application may consist of multiple robots performing assembly and a conveyor belt carrying the part from one robot to the next. In such a case, each robot would be its own subsystem, and integration of the subsystems would be accomplished through the conveyor belt and its corresponding software. As another example, the computing needs of an automobile may be split into subsystems based on the personnel needed to build each. One subsystem may be the fuel injection; another the speed control; a third is the entertainment system, and a fourth is a distributed network that contains processors for all of the power locks and windows.

Definition of a subsystem should not be based on the target hardware. Rather, base each subsystem on a physical or functional entity that can be tested independently. It does not matter whether a subsystem is spread across multiple processors, or a single processor consists of multiple subsystems. When using component-based software, it is easy to move software from one hardware platform to another.

As an example, consider the design of an automotive speed control subsystem. The goal of a cruise control subsystem is to maintain a constant velocity of the car despite the presence of real forces such as friction and gravity. The velocity of the car is continuously monitored. If the car slows down to below the desired speed, more gas is given to speed up the car. If the car speeds up due to going down a hill, the accelerator releases, and if necessary, the brake is applied lightly. The driver must also be able to override the cruise control function, by pressing on the brake or accelerator pedal instead of the buttons that form the cruise control user interface.

There are no precise rules for decomposition. Most engineers rely on prior expertise. New engineers use common sense, but do not necessarily end up with the best design. As a more systematic method of splitting the subsystem into components, the following guidelines are offered. These guidelines can help yield a pretty good first draft of the components in the system. Further refinement is often needed, either by decomposing some blocks further, merging other blocks that are very similar, or modifying the functionality to properly meet all requirements. It is important to understand that in such a design, there is no right answer. However, when comparing two different designs, specific questions can be asked to compare whether one design is better than another; such questions are dispersed throughout the following discussion.

Create one module for each output I/O element

The starting point for any subsystem should be the I/O elements. An I/O element is a sensor, switch, light, actuator, or other application-dependent item, or communication to an external subsystem. An I/O element is not an I/O port, such as serial or parallel port. Rather, an I/O port is encapsu-

lated within a software component of an I/O element.

Draw the I/O elements (on paper, blackboard, or using a computer drawing tool—whichever is preferred) such that all the input elements are on the left, and all the output elements are on the right. Each module should be drawn as a box with rounded corners and the I/O item entering or exiting from the bottom for an input or output module respectively, as shown in Figure 12. Give each module a name that reflects the I/O element.

For example, in a cruise control system, the outputs are the forces to be applied to the engine (how much gas to give), the forces applied to the wheels (how much to press the brake), and output to the dashboard (the speedometer and odometer). These are shown in Figure 13. The remainder of this example builds upon this diagram.

If an I/O element has both input and output, then split the element into two pieces, with input on the left, output on the right. Later in the design, the modules can be merged so that both input and output are implemented as a single component.

Many times, a decision has to be made as to what level of granularity should be made at this step. Suppose there are eight input switches; should this be eight separate I/O modules? Or a single module that has an 8-bit parallel input? As a general rule, if each switch has similar functionality (for example, each switch controls a light) then keep them together. If the switches are used for different functionality (one is for a light, a second turns on a motor, and so on) then split them up. Either way, refinements of the design can be made throughout; it is not important to get the perfect design first try. These guidelines simply provide a starting point.

Communication with another subsystem should also be viewed as an I/O element. For example, the dashboard that contains the speedometer might be a separate subsystem. The speedometer and odometer components of the dashboard, however, are still modeled the same way, because at this point, each component is still independent of the target hardware platform.

FIGURE 14 Revised output elements, showing input ports

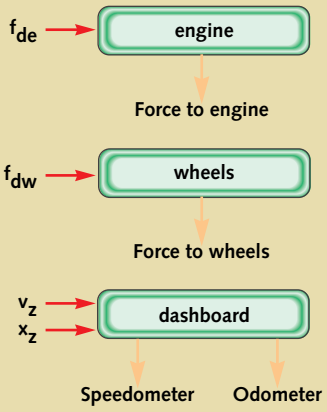
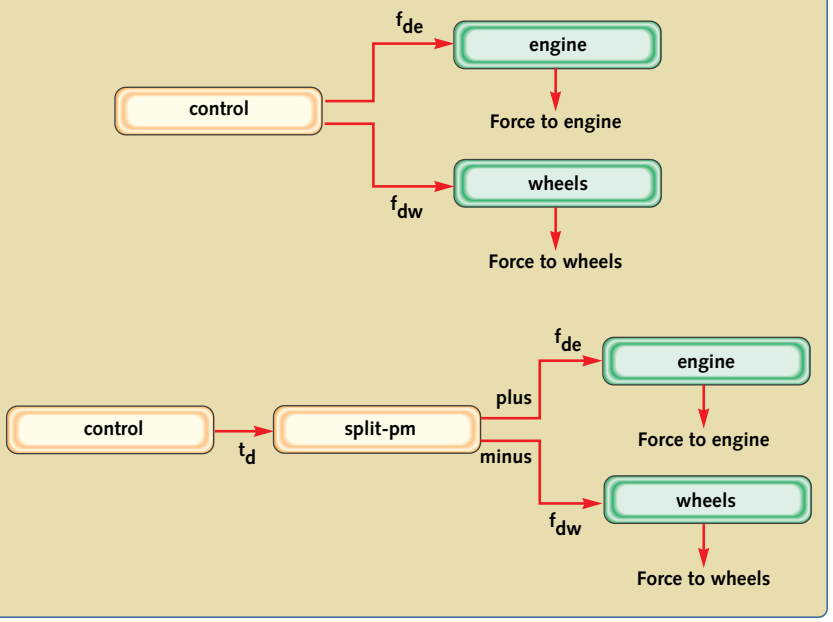


FIGURE 15 Two alternatives for components that generate the force inputs needed by engine and wheel components



Determine inputs to output components

Once all of the I/O components are defined, the diagram is to be filled in from *right to left*, that is, start with the output modules, and work back towards the input modules. To begin, define the input ports of each output module. For the cruise-control example, this means the engine, wheels, and dashboard modules, as shown in Figure 14.

The inputs are variables with standard units that are to be sent or applied to the output I/O element.

FIGURE 16 Modified split-pm module to handle manual overrides

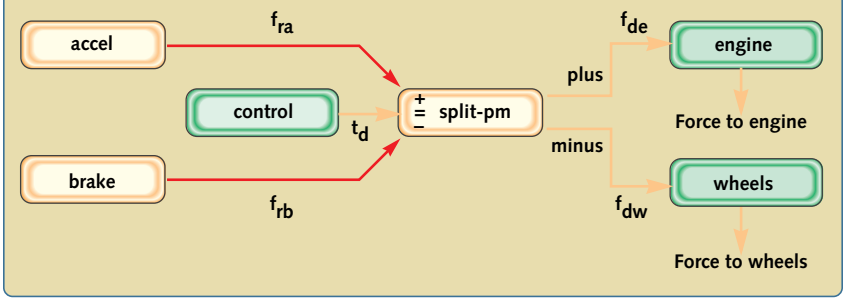
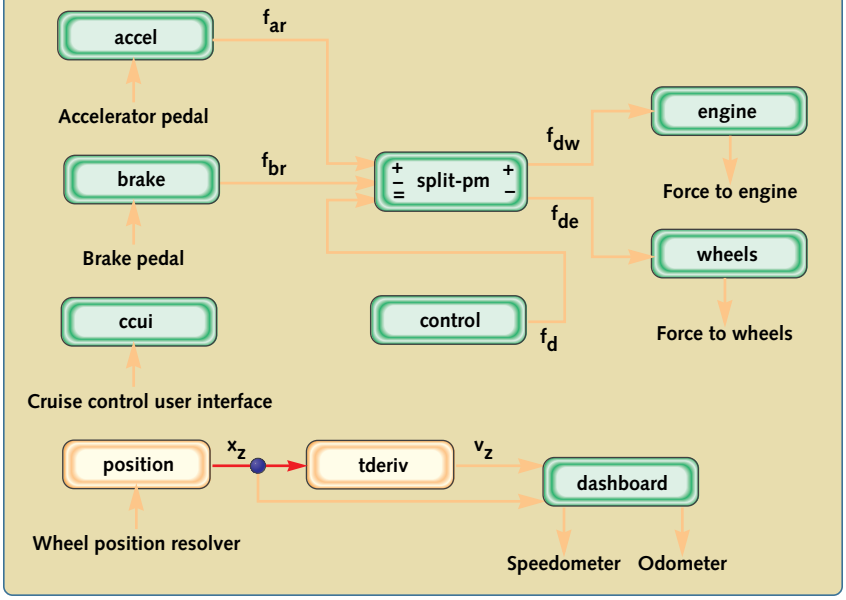


FIGURE 17 Decomposition after ensuring that there is a component producing output that matches the input of each output module



For example, the force for both the engine and wheels should be in Newtons (N), while the input to the speedometer is in meters per second (m/s) or kilometers per hour (km/hr). Of course, the British system of units can also be used; what is important is to be consistent and only use standard units. Do not use raw data values, like a 12-bit value needed by a digital-to-analog converter (DAC). Rather, those conversions will be encapsulated into the I/O components. If performance when using standard units is an issue (for example, floating point vs. integer), address it later during the implementation phase, when the target hardware is taken into consideration. Software decomposition is part of the architectural design phase, and should be independent of the target computing hardware.

By using standard units only, each software component can more easily be used in multiple configurations. Furthermore, it makes it very easy to later revise a display module to display in the user's preferred units of measurement, such as selecting a digital speedometer to display in either km/hr or mph).

In the diagram, use mathematical notation for the variables. For example desired force to be applied to engine can be f_{de}, while force applied to wheels can be f_{dw}. The dashboard inputs may be called measured velocity (v_z) and measured position (x_z). It is important to use formal naming conventions here, because it emphasizes that the most precise way to define a control system is through math. By using these variable names, it is obvious to the control system designer what are the inputs and outputs. Be sure to create a table that matches these variable names to the full names. Also be consistent. The item that matches the units of the variable (that is, force, velocity, and so on) are the variable names, and the specific instances of the variable (that is, desired, measured, and so on) are the subscripts.

Define computational modules

In the same way that a module was created for each I/O output element, now create the modules for each variable needed by one of the output components. For each module, determine what input would be needed, what is the function, and what is its output.

For example, the engine and wheels each need a force that is output by a speed-control algorithm. If the force to be applied to the vehicle is positive, then we want f_{de}, otherwise we want f_{dw}. We can do this in two possible ways: either as a single control module that produces both outputs, or as a control module with a single force output, and a second module that splits the variable depending on whether it's positive or negative. The two possibilities are shown in Figure 15.

Neither design is optimal; rather, there are trade-offs, such that the one that is selected by the designer is based on which criteria in the system needs to be optimized most. The first version results in less modules, and thus may have slightly better performance and use less memory. The exact amount of overhead used by each component is usually small, but not always negligible.[19]

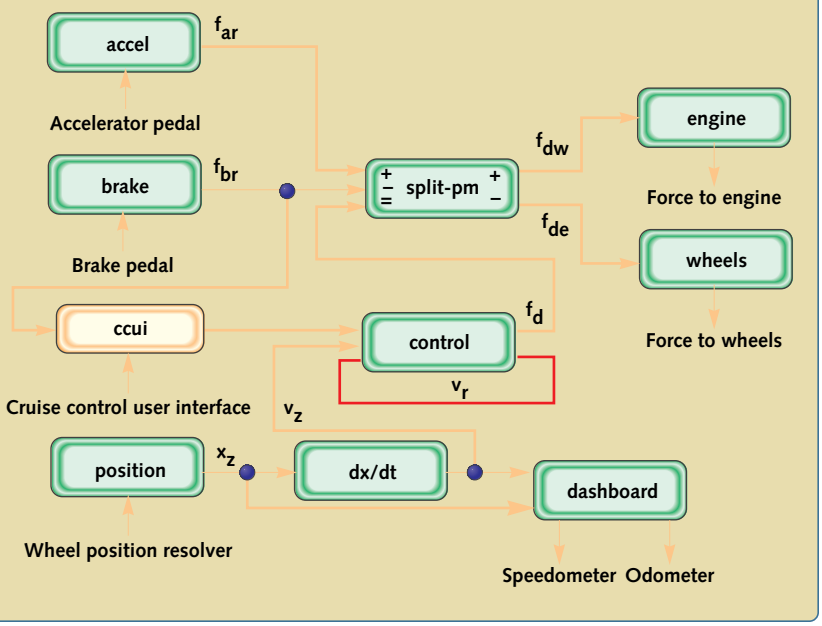
FIGURE 18 Revised control module with input ports shown

The second version, on the other hand, maximizes flexibility of the components. For example, the I/O output that was specified in the structure assumes that acceleration and braking are handled by two different components. What if an electric vehicle with a direct drive motor is used instead of a gas vehicle? The vehicle may be designed such that both acceleration and deceleration have a single force value applied to the motor. In such a case, the *split-pm* module can be removed, with the force f_d directly sent to the I/O output component.

Using the *split-pm* module has an added benefit, as it allows for easily overriding the output of the control module if the driver presses the accelerator or brake, rather than using the cruise-control interface. The modified *split-pm* module is shown in Figure 16 If f_{ra} == f_{rb} == 0, then the input f_d from the control module is used, otherwise f_{ra} and f_{rb} are used. This module demonstrates the situation where the computer is in charge of overriding the engine and brakes. For safety, it may be desirable to allow mechanical overrides which bypass the computer, in which case this code would have no useful purpose and can be omitted.

If a variable is produced by one of the I/O input elements, draw an arrow from the appropriate input element. This is shown for both the *accel* and *brake* modules in Figure 17. The x_z input to the dashboard module is another example; it can be connected directly from the *position* module which would generate x_z as an output port.

FIGURE 19 Complete configuration of software components for cruise control example



The dashboard module also needs measured velocity (v_z). This can be obtained by differentiating the position over time. The measured velocity is derived from the measured position. For that, we can add a time-derivative (*tderiv*) module that takes the input x_z, and produces the output v_z. The *tderiv* module is a component that can be reused often, whenever a time derivative is needed.

The above iteration is then repeated, identifying any module that does not yet have any component producing its input. In this case, only the *control* module remains.

The control module incorporates the algorithm to be used to implement the cruise control. If a simple controller is used, the inputs of the module need to be the reference velocity (v_r), and the current measured velocity (v_z). If v_r== v_z, then the output f_d should be 0. If they are not equal, however, a positive or negative force is output by the control module. The precise value depends on the magnitude of the difference and the gains applied within the control algorithm.

The function of the cruise control algorithm also depends on the commands from the user. For example, the user interface for cruise control is a set of keys (for example, on, off, set, accel, resume, coast), any of which can be pressed by the driver to control the function of the algorithm. For example, if the accel button is pressed, the car must accelerate, and the reference velocity revised accordingly. To handle the commands, the user reference (U_r) variable is also needed as an input to the *control* module. Whereas the other variables were mathematical, U_r is discrete, holding the value of the last pressed button. In this software component framework, there is no difference in the way the code is defined to handle each type of variable, thus the input can be shown the same way as the other variables. For ease of understanding the diagram, however, the

discrete variable is shown as a capital letter. The revised control module is shown in Figure 18.

The output of the control module was already connected. The source for each input variable of this module must now be connected. The input U_r comes directly from the input module *ccui*. Thus that connection is made, and no additional modules are needed for that variable.

The input v_z is already available as the output of the *tderiv* module, so that connection can be made.

The reference velocity is a little trickier. It is not obvious where it comes from. As a design decision, let the control module be responsible for setting it. That is, when the *set* or *accel* button is pressed on the driver's cruise control keypad, the control module sets v_r to v_z. Thus the control module not only has v_r as an input, but also an output feeding back to itself.

Once the first draft of the decomposition is complete, refinements can be made. For example, the requirements for the cruise control specify that if the brake pedal is depressed, the cruise control function is deactivated just as it would be if the driver deactivated it. The designer may choose to have the *ccui* module also monitor the brake pedal. This refinement, with the control module included and fully connected, is shown in Figure 19.

The purpose of this example is to demonstrate decomposition of an application into components; it is not to provide the reader with the best algorithm for doing cruise control. Further modifications can be made as necessary at this stage of the design to satisfy the needs of the application. For example, to obtain better control system performance, it may be desirable to implement a more complex algorithm, in which case additional inputs to the control module are needed. To obtain measured acceleration (a_z), the *tderiv* module can be replicated, with v_z as the input, and a_z as the output.

Applying good technique

Design of real-time software components does not require sophisticated tools. Rather, it requires applying good design techniques as outlined in this article, and discipline to setup an application-independent framework first, then to create software blocks that adhere to the interface specifications of the framework, so they can easily be plugged in and used.

While it is not possible to provide, in this article, all of the details necessary to replicate the design, sufficient detail was provided for the reader to gain an appreciation for what is needed. More details are available in the references.[5,9,15–20] Sample code with more details is also available online.[2]

Although it may take a bit of time to setup the initial framework, once that is done, the framework can be reused over and over again. Many of the software components can also be reused. In turn, the amount of new software that needs to be developed for each new application will decrease, and the software will be much easier to debug and maintain, due to the strict modularity that is a side effect of developing reusable software components. esp

Dave Stewart is executive vice president and chief technology officer of Embedded Research Solutions LLC (www.embedded-zone.com), a consulting and contracting firm. Prior to that, Dave was director of the Software Engineering for Real-Time Systems Laboratory and a faculty member in computer engineering at the University of Maryland. His research has focused on next generation real-time operating system technology and tools to support the rapid design and analysis of component-based real-time software. He has a PhD in computer engineering from Carnegie Mellon University. His e-mail is dstewart@embedded-zone.com

References

1. Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, v.15, n.3, pp. 744–751, 1976.
2. Software Engineering for Real-Time Systems Laboratory, The Echidna Real-Time Operating System, Dept. of Electrical and Computer Engineering, University of Maryland, *www.ece.umd.edu/serts/echidna*.
3. Blake, B.A. and P. Jalics, "An Assessment of Object-oriented Methods and C++," *J. Object-Oriented Programming*, v.9, n. 1, Mar.-Apr. 1996, pp. 42-48.
4. Dorf, R.C. *Modern Control Systems, Third ed*. London: Addison-Wesley, 1980.
5. Hassani, M. and D. Stewart, "A Mechanism for Communicating in Dynamically Reconfigurable Embedded Systems," *Proc. of High Assurance Systems Engineering Workshop*, Washington DC., August 1997.
6. Kelmar, L. and P.K. Khosla, "Automatic Generation of Forward and Inverse Kinematics for a Reconfigurable Modular Manipulator System," *Journal of Robotics Systems*, v.7, n.4, Aug. 1990, pp. 599-619.
7. Liu, C.L. and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *Journal of the ACM*, v.20, n.1, Jan. 1973, pp. 44-61.
8. Molesky, L.D., C. Shen, and G. Zlokapa, "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems," *The Journal of Real-Time Systems*, v.2, n.3, Sept. 1990, pp. 163-180.
9. Moy, M. and D. Stewart, "An engineering approach to determining sampling rates for switches and sensors in real-time systems," *Proc. of Real-Time Applications Symposium*, Washington DC, May 2000.
10. Parnas, D.L., P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.*, v.SE-11, n.3, Mar. 1985, pp. 259-266.
11. Schach, S.R. *Software Engineering, Second ed*. Asken Associates, 1993.
12. Schmitz, D.E., P. K. Khosla, and T. Kanade, "The CMU Reconfigurable Modular Manipulator System," *Proc. of*

the Int'l Symp. and Exposition on Robots (ISIR), Sydney, Australia, pp. 473-488, Nov. 1988.
13. Steenstrup, M., M.A. Arbib, and E.G. Manes, "Port Automata and the Algebra of Concurrent Processes," *Journal of Computer and System Sciences*, v.27, n.1, Aug. 1983, pp. 29-50.
14. Stevens, W.P., G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems Journal*, v.13, n.2, pp. 115-139, 1974.
15. Stewart, D.B. and G.A. Arora, "Dynamically Reconfigurable Embedded Software, Does It Make Sense?" *Proc. Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, Montreal, Canada, pp. 217–220, October 1996.
16. Stewart, D.B. and P.K. Khosla, "Mechanisms for Detecting and Handling Timing Errors," *Comm. the ACM*, v.40, n.1, Jan. 1997, pp. 87–94.
17. Stewart, D.B., D.E. Schmitz, and P.K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-based Control Applications," *IEEE Trans. Systems, Man, and Cybernetics*, v.22, n.6, Nov./Dec. 1992, pp. 1282-1295.
18. Stewart, D.B. and P.K. Khosla, "Chimera 3.1: The Real- Time Operating System for Reconfigurable Sensor-Based Control Systems, Program Documentation, Advanced Manipulators Laboratory," The Robotics Inst. and Dept. Electrical and Computer Eng., Carnegie Mellon Univ., Pittsburgh, *www.ece.umd.edu/serts/bib/manuals/chimera.html*.
19. Stewart, D.B., R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Software Engineering*, v.23, n.12, Dec. 1997.
20. Stewart, D.B., "Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems," doctoral dissertation, Carnegie Mellon Univ., Dept. Electrical and Computer Eng., Pittsburgh, 1994, *www.embedded-zone.com/bib/thesis/dstewart.html*.
21. Wegner, P. "Dimensions of Object-Oriented Programming," *Computer*, v.25, n.10, Oct. 1992, pp. 12–20.
22. Wegner, P. "Concepts and Paradigms of Object-oriented Programming," *OOPS Messenger*, v.1, n.1, Aug. 1990, pp. 7-84.

Endnotes

1. Although the term process is used throughout this paper, implementation in our RTOS is done using lightweight processes, which are also called threads in many operating systems, or jobs if using a non-preemptive real-time executive.