

## 3. Developer's Guide

---

---

Echidna supports the featherweight port-based object (FPBO) model for designing component-based, reconfigurable, real-time software for embedded processors. The FPBO model is for use with a non-preemptive scheduling system. It is based on the port-based object model (PBO), which is for preemptive systems. More information on PBOs and the rationale behind them can be found in [5] .

In this chapter, you will learn how to design and implement a subsystem using reconfigurable software components. First, we will investigate how to decompose your subsystem into components, each with separate functionality. You will learn how to design these components to communicate with each other and how to use techniques for optimizing the reconfigurability and reuse of these components. Then, we will explain how to create software by implementing these components in C using the Echidna FPBO interface. We will also describe other code needed to create a fully working application on Echidna.

Throughout this chapter, we will use an automobile speed control simulator as a sample application to demonstrate these design and implementation techniques.

### 3.1 Decomposition of a Subsystem

---

Let us assume that you would like to create an application to solve a problem with real-time constraints. Some questions you should ask yourself while refining your problem and determining your system requirements include:

- Why do I need to create the application?
- What will the application do?
- What kind of input data will the system need?
- How will the application manipulate the input data?
- What kind of data will the system output?

We will answer these questions as we investigate the cruise control sample application.

### 3.1.1 Define the subsystem

After you have an application in mind, you must divide it into different subsystems based on functionality.

#### subsystem

A *subsystem* is defined as part of an application that can be developed and tested independently, and integrated into an application later through simple communication. For 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.

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

#### component

A subsystem is made of many *components*. Each component encapsulates the functionality of a particular part of the subsystem. To develop component-based software, it is necessary to draw clean boundaries between each component. There are no precise rules for decomposition. Most engineers rely on prior experience. New engineers use common sense, but do not necessarily end up with the best design. Many attempts at creating component-based software fail because designers have no guidelines to follow when breaking up the application into pieces. In the following sections, we present general guidelines for decomposing a subsystem.

For example, in an automotive speed control subsystem, the goal of having a cruise control subsystem is to maintain a constant velocity of the car despite the presence of resistive forces such as friction and gravity. The computer continuously monitors the velocity of the car. 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 computer releases the accelerator, and if necessary, lightly applies the brake. The driver must also be able to override the cruise control function by pressing on the brake or accelerator pedal, instead of having to use the cruise control buttons. By thinking of the requirements for the subsystem, we can convert each requirement into the definition of the parameters for a component.

In Section 3.1.2 to Section 3.1.4, we offer a systematic method of decomposing a subsystem into components. 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 components further, merging other components 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, you can ask specific questions to compare whether one design is better than another; examples of such questions are dispersed throughout the discussion in this chapter.

### 3.1.2 Create a component for each I/O element

#### I/O element

The starting point for decomposing any subsystem is determining the *I/O (input/output) elements*. An I/O element is a sensor, switch, light, actuator, or any other

application-dependent item, including 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 encapsulated *within* a software component of an I/O element.

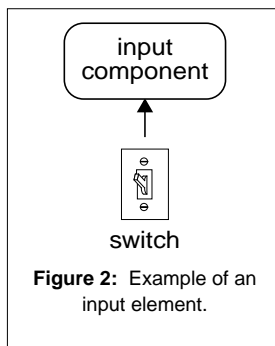


Figure 2: Example of an input element.

An essential tool to use when decomposing a subsystem is a *data flow diagram*. Start creating one by drawing 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 I/O element will be mapped to a component, which should be

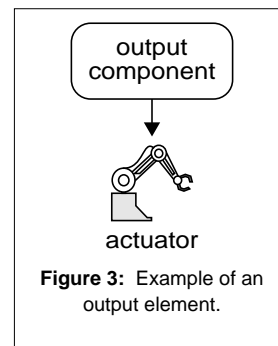


Figure 3: Example of an output element.

properly named to reflect the I/O element. We will represent each component in a box with rounded corners. The I/O element will be drawn below the box with an arrow entering or exiting the box corresponding to an input or an output, respectively. Examples are shown in Figure 2 and Figure 3.

For example, in a cruise control system, the inputs include: (a) the force applied to the accelerator, (b) the force applied to the brake, (c) user input, and (d) the position of the wheels. The outputs include: (e) the forces to be applied to the engine (how much gas to give), (f) the forces applied to the wheels (how much to press the brake), and (g) output to the dashboard (the speedometer and odometer). These are shown in Figure 4.

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 components can be merged so that both input and output are implemented as a single component.

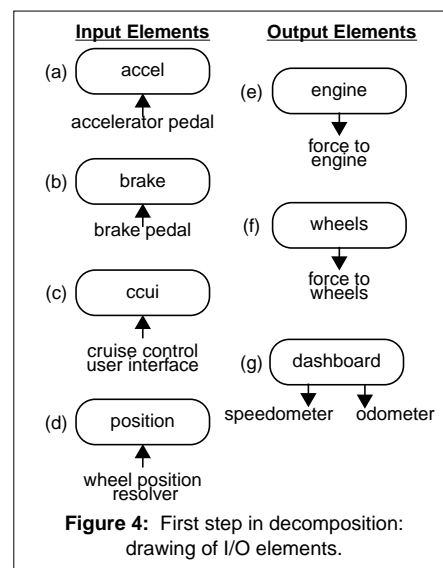


Figure 4: First step in decomposition: drawing of I/O elements.

Many times, a decision has to be made as to what level of granularity should be made at this step. Suppose there are 8 input switches; should this be 8 separate I/O components? Or a single component that has an 8-bit parallel input? As a general rule, if each switch has similar functionality (e.g. 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, etc.) then split them up. Either way, refinements of the design can be made throughout; it is not important to get the perfect design on the 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 modelled the same way, because at this point, each component is still independent of the target hardware platform.

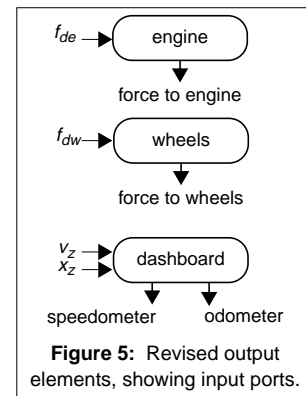
### 3.1.3 Determine the inputs to output components

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 components, and work back towards the input components. To begin, define the input ports of each output component. For the cruise-control example, this means the engine, wheels, and dashboard components.

The inputs are variables with standard units that are to be sent or applied to the output I/O element. 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 within the I/O components. You can address performance issues (e.g. floating point versus integer for standard units) later during the implementation phase, when the target hardware is taken into consideration. Software decomposition is part of the architectural design phase, and it should be independent of the target computing hardware.

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

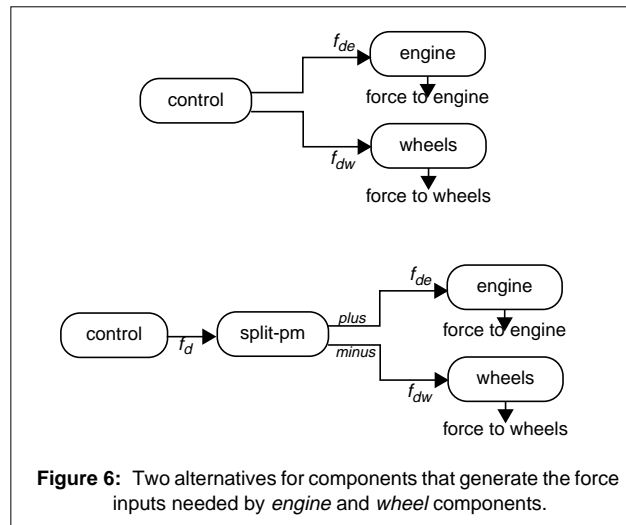
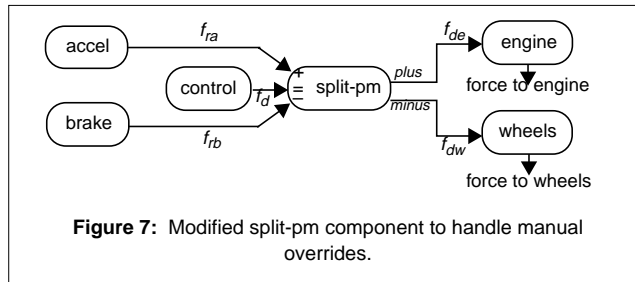
In all of your component diagrams, use mathematical notation for the variables. For example, the desired force to be applied to the engine can be  $f_{de}$ , while the desired force to be applied to the wheels can be  $f_{dw}$ . The dashboard inputs may be called measured velocity ( $v_z$ ) and measured position ( $x_z$ ). Mathematical notation 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 the inputs and outputs are. 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 (i.e. force, velocity, etc.) are the variable names, and the specific instances of the variable (i.e. desired, measured, etc.) are the subscripts. The revised output components are shown in Figure 5.



### 3.1.4 Create the computational components

In the same way that a component was created for each output element, now create a component for each variable needed by each output component. For all components, determine what input is needed, what is its function, and what is its output.

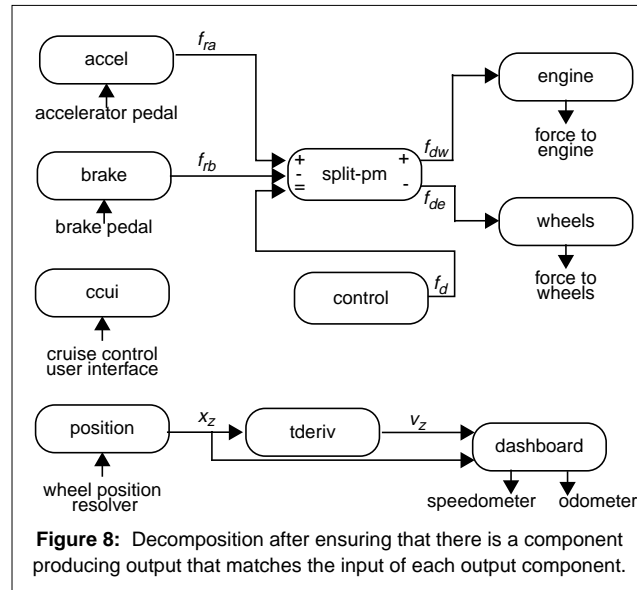
For example, the engine and wheels each need a force that is output by a speed-control algorithm. If the force to be applied to the vehicle is positive, then we want  $f_{de}$  (desired force to engine), otherwise we want  $f_{dw}$  (desired force to wheels). There are two possible ways of generating the inputs for the engine and wheels. Either create a single control component that produces both outputs, or create a control component with a single force output and a second component that splits the variable depending on whether the force is positive or negative. The two possibilities are shown in Figure 6.



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 top version results in less components, 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, as discussed in [5].

The second version, on the other hand, maximizes the 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* component can be removed, with the force  $f_d$  (direct force) sent directly to the I/O output component. If we use this version and decided to change the type of motor later, we can still use the same software.

Using the *split-pm* component has an added benefit, as it allows for easily overriding of the control component output. This would be necessary if the driver presses the accelerator or the brake. The modified *split-pm* component is shown in Figure 7. The force applied to the accelerator pedal is  $f_{ra}$  (reference acceleration) and the force applied to the brake pedal is  $f_{rb}$  (reference deceleration). If  $f_{ra} == f_{rb} == 0$ , then the input  $f_d$  from the control component is used, otherwise  $f_{ra}$  and  $f_{rb}$  are used. This component models the computer's ability to override the engine and brakes. For safety, it may be desirable to allow mechanical overrides that bypass the computer. In that case, the override code would have no useful purpose and could be omitted.



### 3.1.5 Connect input components to computational components

Now, we return to creating the components for each variable needed by the output component. You should continue creating computational components as necessary. If any variables are produced by one of the input elements, then draw an arrow from the appropriate input component to the component you are examining. This is shown for both the *accel* and *brake* components in Figure 7.

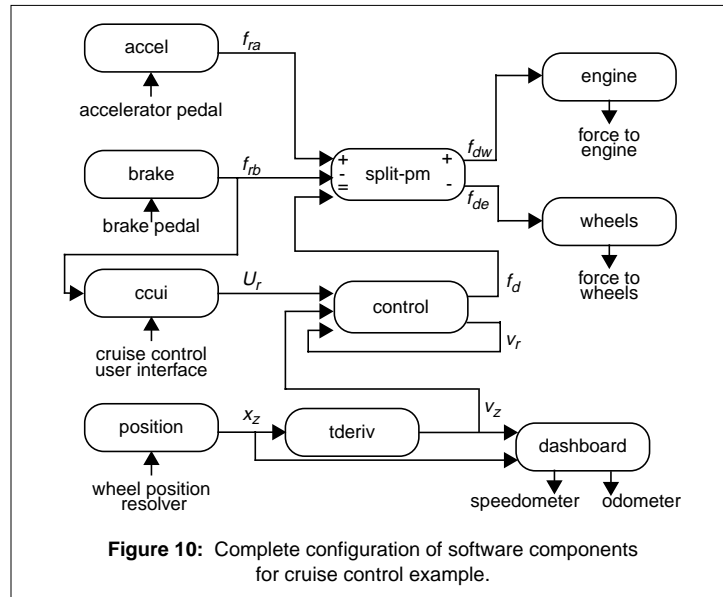
The  $x_z$  input to the dashboard component is another example of data from an input element; it can be connected directly from the *position* component, which would generate  $x_z$  as an output port. The dashboard component also needs a measured velocity ( $v_z$ ). We can obtain  $v_z$  by differentiating the measured position over time. Thus, we can add a time-derivative (*tderiv*) component that takes the input  $x_z$  and produces the output  $v_z$ . The *tderiv* component is an object that can be reused often, whenever a time derivative is needed.

Now, we have components defined to produce the input to each output component. This is shown in Figure 8.

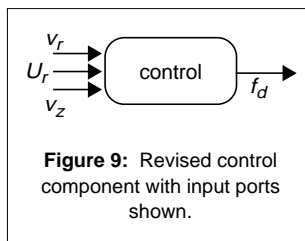
The process of connecting input variables to components is repeated, identifying any computational component that does not yet have any component producing its input. In this case, only the inputs to the *control* component remain to be connected.

The control component incorporates the algorithm to be used to implement the cruise control. If a simple proportional gain controller is used, the inputs of the component should be the reference velocity ( $v_r$ ) that needs to be maintained, and the current measured velocity ( $v_z$ ). If  $v_r = v_z$ , then the direct force output ( $f_d$ ) should be 0. However, if they are not equal, the control component outputs a positive or negative force. The precise value depends on the magnitude of the difference and the value of 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 buttons (e.g. 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 ( $v_r$ ) must be revised accordingly. To handle these commands, the user command or reference ( $U_r$ ) variable is also needed as an input to the *control* component. Whereas the other variables were continuous,  $U_r$  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, 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 component is shown in Figure 9.



The output of the control component was already connected. The source for each input variable of this component must now be connected. The input  $U_r$  comes directly from the input component *ccui* (cruise control user interface). Thus, we draw that connection, and no additional components are needed for that variable. The input  $v_z$  is already available, as the output of the *tderiv* component, so we can draw a line from *tderiv* to the control component.

component.

The reference velocity is a little trickier. It is not obvious where it comes from. As a design decision, we will make the control component responsible for setting the reference velocity. That is, when the *set* or *accel* button is pressed on the driver's cruise control keypad, the control component sets  $v_r$  to  $v_z$ . Thus the control component not only has  $v_r$  as an input, but also an output feeding back into 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 also have the *ccui* component monitor the brake pedal. This refinement, with the control component included and fully connected, is shown in Figure 10.

The purpose of this example is to demonstrate decomposition of a subsystem into components; it is not meant 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 PID (in control systems, this stands for Proportional Integral Derivative) algorithm, which requires additional inputs to the control component. To obtain measured acceleration ( $a_z$ ), the *tderiv* component can be replicated, with  $v_z$  as the input, and  $a_z$  as the output.

### 3.1.6 Summary

When decomposing a system, you must first separate your application into *subsystems*, which are based on functionality. Next, you must divide each subsystem into different components. You will have 3 types of components:

- An *input component* encapsulates an input element like a sensor or switch.
- A *control or computational component* takes data from the input component(s) and uses an algorithm to generate output data for the output component(s).
- An *output component* encapsulates an output element like a light or actuator.

There are 3 basic steps for decomposition of a subsystem:

35. Create a component for each input or output element.
36. Define the computational components.
37. Determine the inputs to the output components.

Remember that each output or input variable for a component must be connected to another component or an I/O element.

Decomposition of a subsystem is the most important and most difficult phase of the design process. The components should be designed for maximum flexibility. For example, if you change one of the I/O elements, you should only have to rewrite the code for one component. If you plan your system well in the design stage, the implementation phase will be quite easy.

## 3.2 Design of the Framework Code

---

In this section, we discuss the framework provided to support the featherweight port-based object model of software components. First, we will present an overview of the system and application modules. Then, we will discuss the communication between application modules and the application programmer interface (API). Finally, we will examine scheduling of tasks in Echidna.

### 3.2.1 System and Application Modules

**module**

A module is a software implementation of a component (e.g., a .c or a .h file in C).

[\(WORK IN PROGRESS SECTION\)](#)

## 3.3 The Framework Process

---

Echidna is created for low end microcontrollers. In low end microcontrollers it is not always possible to map each software component to its own process. Instead, each method of a component executes to completion upon instantiation. The framework schedules the instantiations according to a non-preemptive algorithm.

In a non-preemptive case, there is no concept of a process. Rather, the scheduler maintains a list of fpbo's and keeps track whether the fpbo is ready to execute or it is paused until its next start time. Rather than switching to the context of a fpbo, the scheduler instead just calls the appropriate method. The method, defined as a function, then executes to completion, and control returns to the scheduler.

[\(WORK IN PROGRESS SECTION\)](#)

### **3.3.1 echidna (startup)**

### **3.3.2 main**

### **3.3.3 sync**

### **3.3.4 fpbo**

### **3.3.5 schedule**

### **3.3.6 svar**

### **3.3.7 etime**

### **3.3.8 cpu**

### **3.3.9 ekey**

### **3.3.10 vt**

### **3.3.11 mem**

### **3.3.12 btypes**

### **3.3.13 communication**

### **3.3.14 API**

### **3.3.15 Scheduling in Echidna**

---

## 3.4 Module Implementation

---

### modular decomposition

*Modular decomposition* is a technique for creating applications using discrete, reusable, and reconfigurable software modules. Using this method can help save both time and costs in the design and maintenance of any software application.

At the beginning of this guide, you learned how to decompose a subsystem. In our method of modular decomposition, there is a *one-to-one mapping from components to modules*. Therefore, the name of each component will be the name of the .c file containing the code for that module.

(WORK IN PROGRESS SECTION)

### 3.4.1 main

### 3.4.2 xyz

---

## 3.5 Naming and Style Conventions

---

Creating software without naming and style conventions is equivalent to building homes without building codes. Without conventions, every programmer in an organization does their own thing. Problems arise whenever someone else has to look at the code. For example, suppose the same module is written by two different programmers. The code of one programmer takes 1 hour to understand and verify, while the same code by the other programmer takes 1 day. Using the first version instead of the second is an 800 percent increase in productivity!

The primary factor that affects the readability of code is the presence of naming conventions. If strict naming conventions are followed, simply looking at a symbol quickly tells the reader what the symbol is, where it is defined, and whether it is a variable, constant, macro, function, type, or some other declaration. Such conventions must be posted, just as a legend must appear on a design diagram, so that any reader of the code knows the conventions.

This section describes the naming and style conventions that are enforced in the Software Engineering for Real-Time Systems (SERTS) Laboratory at the University of Maryland and used in the development of Echidna.

### 3.5.1 Naming Conventions

Naming conventions are *extremely* important. Software maintainability is directly related to the use of good naming conventions. By looking at any symbol name, you should immediately be able to distinguish between constants, variables, macros, and functions, as well as whether something is local, global, internal, or external. Consistently using the naming conventions in Table 1 will yield all of this information from simply the symbol name.

**Table 1:** Naming conventions for files, variables, types, and functions in C.

Symbol	Description	Symbol	Description
<i>xyz.h</i>	<b>File</b> containing header info for module <i>xyz</i> . Anything defined in this file <b>MUST</b> have an "xyz" or "XYZ" prefix and be exported by the module.	<i>xyz.c</i>	<b>File</b> containing code for module <i>xyz</i> .
<i>xyz_t</i>	<b>Primary data type</b> for module <i>xyz</i> . Define in <i>xyz.h</i>	<i>Abcde_t</i> <i>AbcdeFgh_t</i>	<b>Internally-defined data type.</b> Define at top of <i>xyz.c</i> . Note that data types and functions for internal use start with a capital letter.
<i>xyzAbcde_t</i> <i>xyzAbcdeFgh_t</i>	<b>Secondary data type</b> "Abcde" for module <i>xyz</i> . Define in <i>xyz.h</i> . All <i>enum</i> 's should be <i>typedef</i> 'd with this format.		
<i>XYZ_ABCDE</i> <i>XYZ_ABCDE_FGH</i>	<b>Constant</b> for module <i>xyz</i> . For <b>internal or external</b> use. Define in <i>xyz.h</i> .	<i>_ABCDE</i> <i>_ABCDE_FGH</i>	<b>Internal constant.</b> Define at top of <i>xyz.c</i> . E.g., if <i>ABCDE_FGH</i> is used instead of <i>_ABCDE_FGH</i> , it implies module <i>abcde</i> .
<i>XYZ_ABCDE()</i>	<b>#define'd macro</b> for module <i>xyz</i> . For <b>internal or external</b> use. Define in <i>xyz.h</i> .	<i>_ABCDE()</i>	<b>#define'd macro</b> for <b>internal</b> use only. Define in <i>xyz.c</i> .
<i>xyz_abcde</i> <i>xyz_abcdeFgh</i>	<b>Exported global variable</b> defined for module <i>xyz</i> . Declare as <i>extern</i> in <i>xyz.h</i> and define in <i>xyz.c</i> . Global variables should be AVOIDED!	<i>_abcde</i> <i>_abcdeFgh</i>	<b>Internal global variable.</b> Define as <i>static</i> at top of <i>xyz.c</i> .
		<i>abcde</i>	<b>Local variable.</b> Define inside a function. Also define fields within a structure using this convention.
<i>xyzAbcde()</i> <i>xyzAbcdeFgh()</i>	<b>Exported function</b> "Abcde" defined in module <i>xyz</i> . Declare as <i>extern</i> in <i>xyz.h</i> and define in <i>xyz.c</i> .	<i>Abcde()</i> <i>AbcdeFgh()</i>	<b>Internal function.</b> Declare prototype as <i>static</i> at top of <i>xyz.c</i> . Define function at bottom of <i>xyz.c</i> after declaring all exported functions.

### 3.5.1.1 Pairing Function Names

You should always name functions such that each exported function has a converse, as shown in Table 2. By defining functions as pairs, there are two important benefits. It forces you, the designer, to ensure completeness. It also allows you to create the two portions simultaneously and use each part to test the other part.

Make sure that pairings are consistent. For example, if the conventions shown in Table 2 are used, make sure the converse of *send* is not *read* and that the converse of *create* is not *finish*. If you are creating the code for reading and writing at the same time, you can test both pieces of code by writing from one process and reading from the

other. It is worthwhile to always use the same conventions for similar components, especially if they are in different modules.

### 3.5.1.2 Compounding Function Names

To allow for further decomposition, put names in “big to small” order for compounded function names instead of in the order that it would naturally be read. For example, if module *xyz* has a secondary structure *xyzFile\_t*, then functions that operate on that structure should be named the following:

```
xyzFileCreate
xyzFileDestroy
xyzFileRead
xyzFileWrite
```

and not

```
xyzCreateFile
xyzDestroyFile
xyzReadFile
xyzWriteFile
```

Note that the last word for any function name should be the *verb* that represents the action performed by the function. The middle words are typically *nouns* to represent the object(s) on which the verbs act.

### 3.5.1.3 Matching names to modules

This convention makes it obvious that *xyzFile* is a sub-structure of the *xyz* module in the first part of the previous example. In the second part, it is not at all obvious. Furthermore, if module *xyz* grows and you decide to further decompose it, then it will be easy to move the entire *xyzFile* sub-structure and corresponding functions to a separate module (e.g., *xyzfile*). A global search-and-replace of *xyzFile* to *xyzfile* would result in all the necessary changes, and within a few minutes, the decomposition is complete. If this naming convention is not followed, it will take much longer to revise all of the names for use in the new module.

### 3.5.1.4 Abbreviating function names

While it is acceptable to have an abbreviated module name because the name serves as a prefix to everything, only use *obvious* abbreviations for function names. If an obvious abbreviation is not available, then use the full name. If an abbreviation is used, then use it *everywhere* in the project.

For example, it can be a convention to always use *xyzInit* as the initialization code for module *xyz*; and never to use *xyzInitialize*. As another example, use either *snd* and *rcv*, or *send* and *receive*, but don't intermix the two. Examples of other common abbreviations include *intr* for interrupt, *fwd* for forward, *rev* for reverse, *sync* for synchronization, *stat* for status, *ctrl* for control.

**Table 2:** Naming conventions for pairing function names.

xyzCreate ↔ xyzDestroy	xyzAlloc ↔ xyzFree	xyzRead ↔ xyzWrite
xyzInit ↔ xyzTerm	xyzOpen ↔ xyzClose	xyzSnd ↔ xyzRcv
xyzStart ↔ xyzFinish	xyzUp ↔ xyzDown	xyzStatus ↔ xyzControl
xyzOn ↔ xyzOff	xyzGo ↔ xyzStop	xyzNext ↔ xyzPrev

On the other hand, an abbreviation like *trfm*, supposedly short for transform, is not recommended, because the abbreviation is not obvious and thus readability is compromised. In such a case, it would be better to choose the function name without abbreviation, *xyzTransform()*. As another example of over-using abbreviations, consider the difference between *xyzFileCreate()* and *xyzFilCrt()*. The second one uses uncommon abbreviations, and it is difficult to follow when reviewing the code during the software maintenance phase. It is much better to use slightly longer names, and avoid confusion as to what the function does.

### 3.5.1.5 Avoiding use of global variables

Global variables are often frowned upon by software engineers, as they violate encapsulation criteria of object-based design and make it more difficult to maintain the software. While those reasons also apply to real-time software development, it is even *more* crucial to avoid the use of global variables in real-time systems.

In most RTOS's, processes are implemented as threads or lightweight processes. Processes share the same address space to minimize the overhead for executing system calls and switching contexts. The side-effect, however, is that a global variable is automatically shared among all processes. Two processes that use the same module with a global variable defined in it will share the same value. Such conflicts will break the functionality. Thus, the issue goes beyond just software maintenance.

Many real-time programmers use global variables to their advantage, as a way of obtaining shared memory. In such a case, however, care must be taken, as any access to it must be guarded as a critical section to prevent problems due to race conditions. Unfortunately, most mechanisms to avoid race conditions (e.g., semaphores), are not real-time friendly, and they can create undesired blocking. The alternatives, such as the priority ceiling protocol, use significant overhead. You can find more information about critical sections and race conditions in any operating systems textbook.

## 3.5.2 Style Conventions

Using naming conventions is only the first step in creating reusable software. You can make your code even easier to read and maintain by following proper guidelines for indentation, spacing, and commenting.

### 3.5.2.1 Indentation

Indentation must always be 4 spaces. Nested and sub-statements must be indented properly.

An example of proper indenting and location of parentheses and braces:

```
void funcname(int a) {
    code goes here
    code goes here
    code goes here
} // end funcname
```

- No space between **funcname** and (
- Always use a space between ) and {
- A comment can be used to show end of function

### 3.5.2.2 if() and if()-else statements

```
if (condition) {
    code goes here
}
```

```
if (condition) {
    code goes here
} else {
    else code goes here
}
```

- Always use a space between **if** and (, or it will look like a function name.
- Always use a space between ) and {

For if-else statements, if you use parentheses on the **if** part, then use it on the **else** part too, and vice versa:

#### Don't write:

```
if (condition) {
    line1;
    line2;
} else
    line3
```

#### Do write:

```
if (condition) {
    line1;
    line2;
} else {
    line3
} // end if-else
```

Similarly:

#### Don't write:

```
if (condition)
    line1;
else {
    line2;
    line3
} // end if-else
```

#### Do write:

```
if (condition){
    line1;
} else {
    line2;
    line3
} // end if-else
```

**Reason:** If you need to add any code later on, it is easy to make a mistake and not add the braces when you add a line of code.

The only time you don't need the braces is with short one-liners. However, having braces can be helpful even then, similar to the example in the **for()** loop below. For anything more than one-liners, always use braces.

For `if()-else if()-else`, always use braces:

```
if (condition1) {
    line1;
} else if (condition2) {
    line2:
    line3;
} else {
    line4;
} // end if-else
```

- A comment can be used at the end of the last `if()`

### 3.5.2.3 while() and do-while() loops

```
while (condition) {
    code goes here
    code goes here
    code goes here
} // end while
```

- Formatting is similar to `if`
- A comment can be used to show the end of the loop

```
do {
    code goes here
    code goes here
    code goes here
} while (condition);
```

- A semi-colon after `)` is necessary.
- Always use a space between `do` and `{`
- Always use a space between `}` and `while`
- Always use a space between `while` and `(`

### 3.5.2.4 for() loops

An example of proper indenting and location of parentheses, braces, and semicolons:

```
for (init;condition;increment) {
    code goes here
    code goes here
}
```

**Always** use braces when nesting loops, for all loops except possibly the innermost loop, even if they are not needed:

**Don't write:**

```
for (i=0;i<10;++i)
    for (j=0;j<10;++j)
        a[i] += j;
```

**Do write:**

```
for (i=0;i<10;++i) {
    for (j=0;j<10;++j)
        a[i] += j;
}
```

**Reason:** Suppose we realize, oops, we forgot to `init a[i]`. Modify code:

```
for (i=0;i<10;++i) {
    a[i]=0;
    for (j=0;j<10;++j)
        a[i] += j;
}
```

Now, we are forced to add the braces. It is safer to just always use braces.

### 3.5.2.5 switch() statements

An example of a `switch` with only one line per `case` statement:

```
switch (value) {
    case 0:    line0; break;
    case 1:    line1; break;
    case 2:    line3; break;
    default:   line3; break;
} // end switch
```

An example of a `switch` with small number of short lines per `case` statement:

```
switch (value) {
    case 0:
        line1;
        line2;
        line3;
        break;

    case 1:
        line4;
        line5;
        line6;
        break;

    default:
        line5;
        line6;
        break;
} // end switch
```

Following is an example of a `switch` with a large number of lines per case, especially when nesting within the `case` statements, or if lines are very long. This method minimizes the white space from indenting many times:

```
switch (value) {
    case 0: {
        line1;
        line2;
        line3;
    } break;

    case 1: {
        line4;
        line5;
        line6;
    } break;

    default: {
        line5;
        line6;
    } break;
} // end switch
```

### 3.5.2.6 Blank Lines

Use blank lines wisely to organize your program into blocks. Think of your code as you would a well-written document: one thought per paragraph, and each paragraph

perhaps containing multiple sentences. In code, there is one thought per 'block', and each block may contain multiple lines of code. Examples of a block include a loop, a complete if-then statement, or a set of assignments that belong together.

For example:

```
int main(void) {
    int a,b,c;
    float y,z;
                                blank line
    while (condition) {
        statement1;
        statement2;
        statement3;
    }
                                blank line
    // comment belongs to block, so no blank line
    // between it and the code.
    for (...) {
        statement4;
        statement5;
    }
                                blank line
    if (condition) {
        code;
    } else {
        more code;
    }
                                blank line
    return 0;
}
```

Too many blank lines cause your program to span too many pages when you print it out or require extra scrolling when editing. Not enough blank lines makes it difficult to read the code.

### 3.5.2.7 Commenting

You can use either the `/* */` or the `//` method. The `//` method is usually much easier to type and is highly recommended.

Comments should expose the *thought process* behind what a line or block of code is doing, rather than repeat what the code is doing. Saying exactly what the code is doing is often redundant. For example, here are extreme cases of bad comments that are seen very often:

```
count+=2;    // increment counter by 2.
return (sum); // Return the sum
```

Better comments:

```
count+=2;    // increment by 2 because we only want even numbers
return(sum); // sum could be negative if error in input data
```

If a comment would be redundant, and there is really no "thought" that needs to go behind it, then don't comment it. Useless comments are worse than no comments.

Comments are always indented with respect to the code, so that the left-most column is the code. This is accomplished in either of two ways:

- Always start at column 40 (or at column 32. Just make sure it starts at the same column throughout the code), so that your code looks like two columns.
- The left column is code, the right column is comments.

**E.g. Bad commenting:**

```
// comment goes here in Column 8, but code in column 12
  abc = def;
// the problem is that the code gets lost.
```

**E.g. Good commenting:**

```
abc = def;// comment goes here in Column 40

    // long comment can go here, indented immediately before the lines
    // of code to which it pertains.
def = pqr+uvw;
```

When a function, loop, or if-else statement is more than 10 or so lines, or you have multiple levels of indentation, put a `// end xyz` comment after the closing brace. For example:

```
void funcname(int mmax, int nmax, int pmax) {
  int m,n,p;

  for (m=0;m<mmax;+m) {

    for (n=0;n<nmax;+n) {
      for (p=0;p<pmax;+p) {
        code goes here
      } // end for p
      more code goes here
      more code goes here

    } // end for n
    more code goes here

  } // end for m

  more code goes here

  return;
} // end funcname
```

Define code in “paragraphs.” That is, each thought is single-spaced (like a small loop), and double-spaced between different “thoughts”. Put a comment to begin each paragraph that explains what the block of code is doing in high-level terms.

**Example:**

```
static void _sample_function (void) {
  // single space variable declarations
  // double space after them.
  int i,j;
  int a[N],b[N];

  // single space here
  for (i=0;i<N;++i)
    a[i]=i;

  // then double space before next thing, but single space
  // the “paragraph”.
  if (b[i] < a[i]) {
    // you can put a comment here
    printf(“b[i] is smaller”);
  } else {
    printf(“b[i] is bigger”); // or put a comment here
  }

  return;
}
```

Note the `if()-else` bracket and indenting. Another example:

```
// comment for entire if-else structure
if (condition1) {
    // comment for condition 1
    code for condition 1
    code for condition 1
    code for condition 1
} else if (condition 2) {
    // comment for condition 2
    code for condition 2
    code for condition 2
    code for condition 2
} else {
    // comment for else part
    code for else part
    code for else part
    code for else part
}
```

### 3.5.2.8 Expressions and Conditions

Spaces in equations and conditions are optional. Use them to help you quickly see the order of operations. You can also add parentheses, even if they are not needed.

Examples:

Hard to read:

```
x = 4 * y + 3 * z / 2;
x=4*y+3*z/2;
```

Easier to read:

```
x = 4*y + 3*z/2;
```

Hard to read:

```
if ( x == 3 * z + 4&& y == 2 - z ) {
```

Easier to read:

```
if ( ( x == 3*z + 4 ) && ( y == 2-z ) ) {
```

If you have really long conditions, put one condition per line, or format it in such a way that makes sense. E.g.:

```
if ( (long-condition-one) &&
    (long-condition-two) ||
    (long-condition-three) ) {
```

---

## 3.6 Conclusion

Design of real-time software components does not require sophisticated tools. In Echidna, we have set up an application-independent framework. All you have to do is to create software components that adhere to the specifications of the FPBO interface. These modules can easily be plugged in and reused. Thus, 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.

You should now be able to build your own application for Echidna. The template files and sample application files are given in the archived echidna.1.0 file. Along with the source distribution of Echidna, these files will help you to get started. Enjoy!

## 3.7 Notes

---

### 3.7.1 main

*main* is probably the easiest module to write. We must not confuse the *main* module with the `main()` function present in any C program. *main* is the starting point of your application. This is the module that Echidna sets as the first process to run. Main is where you define, create, and destroy all other FPBO.

(WORK IN PROGRESS SECTION)

#### Main and other modules.

All modules, including *main*, share the same template. The source code for each module will contain n sections. xyz can be replaced by the module name, and you can insert the appropriate code into each section

1. **include files:** Header files containing the functions used by the module are included in this section.
2. **macro definitions:** Any macros used by the module are included in this section.
3. **xyzLocal\_t:** In this data structure we declare local variables that are used by any function in the module, and pointers to state variables (variables in the Svar table that are used by other modules).
4. **internal function prototypes:** Functions *other* than the “standard functions” mentioned next should be prototyped in this section.
5. **standard functions:** The module is declared as part of the application in this section. Unlike regular C applications, there are no main() functions in any of the modules when building reconfigurable software modules. The standard functions are as follows:
  - **xyzInit:** Echidna calls this function when the module is first loaded and initialized.
  - **xyzOn:** Echidna calls this function each time the user turns on the module.
  - **xyzCycle:** If user has turned on the module, Echidna executes this function at a frequency specified by the user in the .rmod module configuration file. This is the main loop of the module.
  - **xyzOff:** Echidna calls this function each time the user turns off the module.
  - **xyzTerm:** Echidna calls this function when the program is shut down. This function is responsible for freeing up any resources used by the module.

The parameters passed into the Init, On, Cycle, Off, and Term functions of each module enable the functions to access the local pointers to the state variables. This will be clearer with an example.

(WORK IN PROGRESS SECTION)

## 3.7.2 Compiling and Executing

### 3.7.2.1 Compiling

After you have written the code for main.c and at least one module, you can compile the program.

(WORK IN PROGRESS SECTION)

### 3.7.2.2 Executing

#### How it works

The cycle function of each module is a process that runs periodically on the real-time operating system. It uses the latest data available in the state variable table for its cycle. There is a deadline for each process, and the operating system should execute them according to an internal scheduling algorithm. You do not need to worry about the scheduling. However, if the specified period of a process is shorter than its execution time, the process will start to miss deadlines, and your program will not function as desired. You do have to worry if your cycle time is too short or if your function is too long and complicated to run quickly enough.

(WORK IN PROGRESS SECTION)

#### Common errors

Common semantic and initialization errors which can give erroneous run-time results include:

- Incorrect use of pointers for SVARs (make sure you reference the pointers in the Local\_t structure correctly).
- Spawning the modules in the wrong order such that the state variables are not initialized correctly.
- Running the modules at too high or too low of a frequency (e.g. do not try to print 500 statements per second, or you will crash the system!)

(WORK IN PROGRESS SECTION)

