Writing a Custom Add-On Code Generator for Polled State Machine Applications

Previous topics in this column dealt with low-level applications, accompanied by assembly language code. This month we deviate from this trend and present code written entirely in C. We stick to our practice, however, of dealing with multi-facetted applications. In this case, we look at one of the basic programming styles, namely polled state machines, and experiment with writing our own add-on code generator to automatically produce skeletal programs. The concepts are illustrated by a simple example, and as always, the source code for both the example and the code generator are available for downloading (see [1]). We hope that the topic will be beneficial to those who would like to write their own add-on code generators as well as those who may consider writing their next application in the structure of a polled state machine.

Software Reusability

Code generators are a recent effort in the quest for code reusability. There have always been ways to reuse software. Perhaps the most basic form of software reusability is when a programmer develops a personal style or approach to software development. But such personal experience is not easily transferred from one programmer to the next. Another common form of software reusability is the time-honored "cut-and-paste" approach. Here, code segments are copied from an existing application. "Cut-and-paste" often requires some amount of code modification, which leads to errors. Similarly, modular structured programming promotes placing the unit operations in distinct standard functions. For instance, a library may contain functions to operate the analog-to-digital converter. Modern compiler technologies acknowledge this practice and include linkers and library utilities as fundamental components. Many programmers and institutions develop their own set of functions and package them as libraries. There has been strong third-party support, especially for the PC platform, to develop libraries. Using library functions is appropriate especially when there are only a few ways of doing a set of standard tasks. Consider, for example, a library for RS-232 communications. The library must support half-duplex and full-duplex modes, or say, for hardware handshaking (DTR/CTS) versus software handshaking (XON/XOFF). You may develop (or buy) libraries with configurable functions. That is, the initialization function takes parameters that describe the various operating modes. This approach is convenient for the programmer, since there are only a few powerful configurable functions to learn. On the other hand, these functions are usually somewhat larger than needed since they include code to support all possible operating modes. The alternative is to write separate functions to support the different operating modes. This produces tight code, but is somewhat more difficult for the programmer who now has to study and understand a large set of functions so that the most appropriate function may be put to use.

Code generators present a good approach to create specific functions for user-specified operating modes. This approach combines the best of both worlds: similar to setting function parameters to select the operating modes, the programmer selects the options from the code generator. The programmer need not learn all the possible functions that may be produced. He simply needs to follow the code generator dialogs and select the desired options. Then the code generator produces the streamlined code. This approach does provide the best of both worlds -- but at a higher cost to the developer of the library. Instead of writing (and often distributing source code) of a set of functions, the library developer must produce a user interface to extract the settings from the programmer. Then, he must program enough smarts to produce the desired code. This is probably why there are many more libraries than code generators. Code generators for embedded controllers have usually been concerned with initialization and register configuration. A code generator is doubtlessly more efficient and accurate in selecting the various special-function

register bits, for example, to initialize a peripheral. Siemens' DAvE [2] is a good example of such a code generator. Although somewhat more involved than writing libraries, it is nevertheless possible to write custom code generators. As mentioned, programmers tend to develop their personal style over time. Often, this style reflects more than preferences in esthetics or syntactic idiosyncrasies. It extends to the way the programmer approaches a given application. A custom code generator may capture the programmer's approach in a powerful way. Parallel to the proliferation of libraries, perhaps future programmers or institutions will be more involved in writing their own code generators to ensure uniformity across their products.

The beta release of Rigel Corporation's Reads166 V3.10 allows users to write and install their own code generators into the integrated development environment (IDE). This provides a good opportunity to try some of the concepts we mentioned. Specifically, we will develop a code generator to implement a well-known software architecture, namely a polled state machine.

State Machines as a Software Architecture

We use the term "software architecture" to denote the essence of a software application, those structural features that remain when the specifics of the application are stripped away. Although there are countless embedded control applications, the software architectures these applications adopt comprise a much smaller set. This statement is even more valid if you view all applications developed by the same programmer or same institution. More mechanically, we view the software architecture as a generic framework that could be decorated with the specifics of a given application. We would like to automate the generation of this framework for a given application. Our effort is similar to developing a template or style sheet commonly used by word processors and presentation managers.

All embedded control applications share some common structural features. They all receive inputs from their environment, process these inputs, and generate the appropriate outputs. Most of the time there is a sequence of operations to be performed. It is not uncommon for this sequence to be conditioned on some input. Hence, such a sequence is better depicted by a flowchart with conditional branches rather than a simple list. Similarly, many operations have time constraints. That is, a given operation must be completed in a specified time, or the controller needs to take alternate action. If these times are very short, an interrupt-oriented architecture is perhaps the best approach. On the other hand, if these times are long (in the order of hundreds of milliseconds), the application program may simply cycle through, or poll, the inputs and produce the desired outputs. We assume that our application is suitable for the polled approach. In general, the outputs depend on the inputs as well as the current position in the sequence of operations.

We further formalize the idea of a flowchart and introduce a state machine. Mathematically speaking, a state machine is a process based on a directed graph. The graph consists of a set of nodes and a set of directed arcs that connect the nodes. Each node represents a state of the system. The set of states is referred to as the state space. We use the term "transition" to refer to a change in state. The sequence of operations is modeled by a process of transitions among the states. The transitions are assumed instantaneous. The system may spend time only in states. That is, we adopt the so-called "activity at node" model rather than an "activity on arc" model. For sake of generality and flexibility, we partition the state activities into three categories: entry activities, exit activities, and stay-in-state (residence) activities. Entry and exit activities are performed when the process first enters or finally leaves the given state. The stay-in-state activities are repeatedly performed while the process remains in a given state. Formalizing the concept of states and transitions now allows us to condition the transitions only on the inputs. This view is not only generic enough to be used in modeling a wide range of applications, but also presents a convenient format for writing code.

In fact, the state machine has been a popular architecture not only as a software architecture, but also in capturing user input in hardware design. Almost all CPLD (Complex Programmable Logic Device) and FPGA (Field Programmable Gate Array) development tools, for example, allow the user to define the functionality of the device by a state machine. (For an example, see Xilinx, Inc. development tools [3]). There have been a few attempts to apply the state machine model to general software development. A few years ago, I experimented with the product BetterState by R-Active, which generated C++ and Visual Basic code. (BetterState is now supported by Integrated Systems, Inc. A light version of BetterState may be downloaded off their website [4]). These products have a graphical user interface where the states and the transitions are drawn on screen. Then, by clicking on the states or transitions, application-specific information is input. Our attempt to generating state machine code is much more modest. We limit our code to polling, in a manner similar to most Programmable Logic Controllers (PLCs). Furthermore, we will simply generate source code in the basic structure of a state machine. We will place comments, such as "enter your code here…" to instruct the user to add application-specific code. In this sense, our generator produces only a framework. Except for the state names, no application-specific code is generated.

We implement the state machine in C. Each polling cycle starts by scanning the inputs. The next state is determined by the current state and the inputs. Conceptually, we use two variables, say *sCurrent* for the current state and *sNext* for the next state. The code that determines the next state will typically be a list of conditional statements. If the choice of the next state depends more on the inputs rather than the current state, we may adopt an input-oriented scan.

```
if(input01) sNext=S01;
 else if(input02) sNext=S02;
 else if(input03) sNext=S03;
 else sNext=S04;
```

On the other hand, if the choice of the next state depends more on the current state, we would adopt a current-state-oriented scan. Consider, for example, a simple keypad that has an arrow key that displays the next choice. Let the four states STATE00 through STATE03 correspond to the four menu items and input00 correspond to a signal from the arrow key.

```
 switch(sCurrent)
  {
   case STATE00 : if(input00) sNext=STATE01; break;
   case STATE01 : if(input00) sNext=STATE02; break;
   case STATE02 : if(input00) sNext=STATE03; break;
   case STATE03 : if(input00) sNext=STATE00; break;
   }
```

Of course, in most cases, the conditions would be a bit more involved, involving many inputs. Similarly, the next state would probably be dependent on both the current state and the inputs. Consider the same example but with an additional pushbutton. Let the menu items corresponding to STATE00 and STATE01 be grouped as submenu 1, and the other two as submenu 2. Let the new pushbutton, represented by input01 toggle between the submenus. The following code segment, which illustrates the hybrid structure, may be used to determine the next state.

```
 switch(sCurrent)
  {
   case STATE00 : if(input01) sNext=STATE02;
```

```
                    else if(input00) sNext=STATE01;
                   break;
   case STATE01 : if(input01) sNext=STATE03;
                    else if(input00) sNext=STATE00;
                   break;


   case STATE02 : if(input01) sNext=STATE00;
                    else if(input00) sNext=STATE03;
                   break;
   case STATE03 : if(input01) sNext=STATE01;
                    else if(input00) sNext=STATE02;
                   break;
   }
```

Each implementation eventually will carry out application-specific actions. Writing subroutines for each specific action or activity is a good way to organize the software. Conceptually, these activity functions are similar to interrupt service routines or event handler routines. The state-machine code generator will produce skeletal code into which calls to the activity functions must be inserted. In our model, activities take place at the nodes, or states. We introduce three standard functions to be a part of the general framework: EnterState(), ExitState(), and StayInState(). Each of these three standard functions takes a single parameter, which defines the associated state. When a transition occurs, the state machine calls the function ExitState with its parameter set to the current state. Then the function EnterState is called with its parameter set to the next state. The polling cycle then ends with updating the current state to the next state. If the state does not change during the polling cycle, i.e., if the next state is the same as the current state, we call the function StayInState with its parameter set to the current state. These three functions allow us to neatly place initialization and termination code in a well-defined and well-organized manner. Note that it was a similar need that gave rise to the constructors and destructors in C++ classes.

Writing the Code Generator

The code generator is written in Microsoft Visual C++ following the guidelines given in Reads166 v3.10. In summary, when a C code wizard is dragged-and-dropped into a project, the integrated development environment (IDE) calls the Reads166 code generator. This generator has a “Setup” button that allows users to insert other code generators into the existing list. For a custom code generator to be recognized, it must be written as a Windows Dynamic Link Library (DLL). Moreover, the DLL must contain the following interface functions:

```
// prototypes of common functions
extern "C"
 {
  __declspec(dllexport) int   IsRcgCWizard(void);
  __declspec(dllexport) int   GetNumberOfCodeGenerators(void);

  __declspec(dllexport) const char * GetNthMenuItemName(int n);
  __declspec(dllexport) const char * GetNthFunctionName(int n);
  }
```

These functions are defined as C functions (extern “C”) so that code generators written with other languages may easily be interfaced with the IDE. Reads166 refers to the DLL as a “wizard” and allows each wizard to have the capability to generate different types of code segments, say for timers or state machine templates. A code generator is the conceptual entity that generates a certain type of code. Each code generator is identified by its own name in the IDE dialogs. Thus,

each user-developed wizard may include several code generators. Inserting a user-developed code generator begins with selecting the DLL. The IDE loads this DLL and calls the first function IsRcgCWizard(). If this function exists in the DLL and returns a nonzero value (TRUE) the DLL is assumed to be a valid code generator. Incidentally, you may also develop assembly language code generators. In this case, the function IsRcgAsmWizard() is sought. Provided that the DLL is recognized as a valid wizard, the IDE calls the function GetNumberOfCodeGenerators() and expects an integer count. Each code generator has a menu item name and a function name. The menu item name is retrieved by calling the function GetNthMenuItem() with the parameter n set to the ordinal value between one and the number of code generators previously obtained. The menu items are displayed in the IDE dialog list boxes. Similarly, the IDE calls the function GetNthFunctionName() with the ordinal value as the parameter. The function name is used in invoking the associated code generator of the wizard. If any of these functions fail, the IDE assumes the wizard is not a valid one and aborts the setup.

When the user-developed code generator is to be used, the programmer selects it from the list box by clicking on its menu name. The IDE then calls the function, using the function name retrieved by GetNthFunctionName() during the setup. The function prototype for each code generator is shown below.

```
extern "C"
__declspec(dllexport) int FunctionName(int   nhWin,
                                       char *szPath,
                                       int   nProcessor);
```

Again, the functions are defined as C functions for better portability. The name of the function must match that returned by GetNthFunctionName(). The function is expected to return a success code, zero (FALSE) if there was an error or if the user cancelled the operation, or nonzero (TRUE) if the code generation was successful. The three arguments are provided by the IDE. The first is the handle to the main window of the IDE. The code generator may use this window as its parent window. The second parameter is the path of the file to be generated by the code generator. This argument points to a buffer large enough to hold the largest path. The code generator should ignore any filename or extension in the given path. It should also modify the path by appending the file name and extension of the generated module. It is this filename and extension that the IDE uses to name the new project module. The last argument is the C166 processor core type, currently 166 or 167. The code generator may present its own dialogs, obtain user information and generate one or more source files.

The interface between the Reads166 IDE and user-written code generators contains a few more aspects and features we will skip. If you would like to write your own code generator, you may reuse most of the interface code we develop in this example without modification (see 1). While the interface code remains practically unchanged, the user must provide specific code to produce the particular source for his code generator. During the experiments, I tried two approaches. First, the code to be produced may be viewed as a fixed file with special embedded macros. I used the format %n% where n is a number identifying the macro and enumerated the macros from 1 to N, where N is the total number of variable fields. For example, we may write the source line

```
#define STCOUNT          %1%
```

and save it as fixed text. The code generator gets user information, such as the number of software timer needed. Suppose the user specifies eight software timers. We store this value as

the value of macro 1. Before generating the actual code, our code may scan the fixed text and replace all macros by their values, in this case, %1% by 8 to produce the following final source line.

```
#define STCOUNT          8
```

Note that the macros need not be replaced only by numerical values. For instance, macros may be replaced by user-given variable names or configuration bytes.

As an alternative, the final code may be generated line by line using “printf”-type (or, in C++, << type) statements. The user information is saved by the dialog, say as the dialog object’s data field m_nNumberOfSoftwareTimers. When it is time to generate the line, we incorporate this information in our “printf” statement.

```
printf(sz, “#define STCOUNT %d\n”, m_nNumberOfSoftwareTimers);
```

This line is then added to the output file. Although the first approach sounds more formal, we chose the latter approach due to its simplicity. The dialog code and “printf”-type code generation statements are relatively straightforward in nature, and thus are not listed here.

We will write two code generators, one to implement a set of software timers, and another to implement a polled state machine.

Software Timers Code Generator

Software timers are used in many states for timeout purposes. Moreover, we may need a multitude of software timers, each keeping track of a different timeout, but all operating concurrently. Our code generator produces two files, SoftwareTimers.c and a corresponding header file Stimers.h. The header file contains macros, which may be used by the calling routines. Thus, the header must be included in all such calling modules. The header is entirely fixed, except for the number of software timers, as discussed above.

```
#ifndef STIMERS_H
#define STIMERS_H

#ifndef TRUE
#define TRUE   1
#endif

#ifndef FALSE
#define FALSE  0
#endif

#define EXPIRED(n)  (ST[n].uExpired)
#define ENABLE(n)   {ST[n].uEnable=1;}
#define DISABLE(n)  {ST[n].uEnable=0;}

// --- constants ---
#define STCOUNT          8    // number of software timers

predef unsigned int T0, T0REL, T01CON, T0IC;
predef unsigned bit T0R, IEN;

// --- software timers ---
```

```
struct SoftwareTimer
 {
  unsigned int      uEnable,    // enable flag
                    uExpired;   // becomes TRUE upon timeout
  unsigned long int ulTimeout;  // milliseconds to timeout
  };
```

Each software timer is implemented as a structure. The first two fields are Boolean values. The field uEnable is TRUE if the software timer is in use. When uEnable is FALSE, the software timer is considered off-line. The first field is set to TRUE or FALSE by the code that uses the timer. The second field, uExpired, becomes TRUE upon a timeout. The calling code should not modify this field, only read it. The last field is an unsigned long integer showing the remaining number of milliseconds to timeout. As this implies, the timers as count-down timers. Timeouts from one millisecond to 0xFFFFFFFF milliseconds, or roughly 1193 hours, are possible. Each millisecond, the ulTimeout field of each enabled timer is decremented. Those timers whose ulTimeout reach 0 are disabled and their uExpired fields are set to TRUE.

The macros EXPIRED, ENABLE, and DISABLE simplify accessing the structure fields. The following code, for example, may be employed if a message is to be displayed when timer 3 expires (reaches zero).

```
if(EXPIRED(3)) SendStr("Timeout !\n");
```

Software timers are based on one of the processor's hardware timers. The hardware timer is run in a reload mode. The reload value depends on the CPU frequency. Interrupts are generated every millisecond. The interrupt service routine decrements the ulTimeout field of each enabled software timer. If any of these ulTimeout values reach zero, the interrupt service routine also sets the corresponding uExpired field to TRUE and sets the corresponding uEnabled to FALSE. The C file generated by the code generator contains a global variable, named ST is defined to hold all timer structures.

```
// --- global variables ---
struct SoftwareTimer ST[STCOUNT];
```

ST is an array of STCOUNT elements, each a structure of type SoftwareTimer. The macro STCOUNT is defined in the header file, based on the user input to the code generator, as explained above. The header file also includes macros to access the fields of the software timer structures as shown below. The header file is shown below. Note that although the code is written in C, it takes an object-oriented approach to software timers. Each element of the global variable ST encapsulates a timer. Moreover, the macros EXPIRED(), ENABLE(), and DISABLE() play the role of member functions, but without the overhead of C++ classes.

The generated source file "SoftwareTimers.c " contains four functions: the initialization routines StInit() and ResetSoftwareTimers(), the routine StartSTimer() to launch a timer, and the interrupt service routine TnIsr() that updates the software timers every millisecond. The name of the interrupt service routine depends on the hardware timer selected by the user. User inputs are received by a single dialog box as shown below.

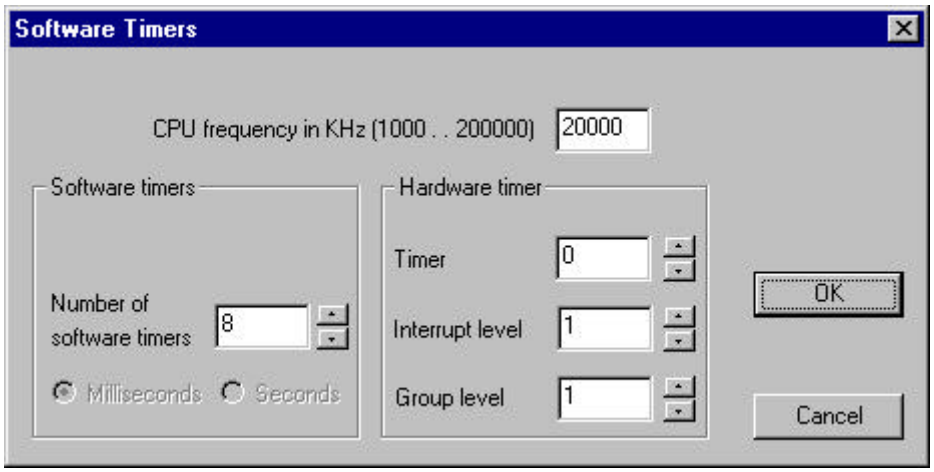

Figure 1. Capturing user information to generate code for software timers.

These settings generate the following StInit routine.

```
void StInit(void){

 T0REL=0xF63C;          // reload value determines the period
 T0=0xF63C;             // start timer with reload value

// --- set up the timer control register ---
 T01CON&=0xFF00;        // prescalar=8, Timer0 off
 T0IC=0x45;             // enable T0 interrupts, level=1, group=1

// --- clear and initialize all timers ------------
 ResetSoftwareTimers();

// --- start timer ------------
 T0R=1;                 // start timer
}
```

StInit() is to be called only once at the beginning of the main program. The other initialization routine ResetSoftwareTimers() are actually called in StInit(). This routine simply resets all software timer fields as shown below.

```
void ResetSoftwareTimers(void){
int n;
 for(n=0; n<STCOUNT; n++)
  {
   ST[n].uEnable=FALSE;
   ST[n].ulTimeout=1;
   ST[n].uExpired=FALSE;
   }
}
```

Note that ResetSoftwareTimers() does not contain any values or fields that depend on the user selections. It is simply generated as fixed text. ResetSoftwareTimers() may be called by the application. It simply has the effect of disabling all software timers.

The routine StartSTimer() is also void of any user-defined values, hence, is generated as fixed static text.

```
void StartSTimer(unsigned int nSTimer, unsigned long int ulMS){

 if(nSTimer >= STCOUNT) return;
 IEN = 0; // suspend interrupts while enabling software timers
 ST[nSTimer].uEnable=FALSE;
 ST[nSTimer].ulTimeout=ulMS;
 ST[nSTimer].uExpired=FALSE;
 ST[nSTimer].uEnable=TRUE;
 IEN = 1;
}
```

The first argument of StartSTimer() specifies the software timer, and the second, the timeout value in milliseconds. The interrupts are monetarily suspended during setting up the timer.

The last function is the interrupt service routine. It is given below for Timer T0.

```
void interrupt(0x80) T0Isr(void){
int  n;

 for(n=0; n<STCOUNT; n++)
  {
   if(ST[n].uEnable)
    {
     if(ST[n].ulTimeout>0) ST[n].ulTimeout--;
      else
       {
        ST[n].uExpired=TRUE;        // set the timer expired flag
        ST[n].uEnable=FALSE;        // and disable the timer
        }
     }
   }
}
```

Other than the interrupt vector, the interrupt service routine is fixed text.

State Machine Code Generator and an Example
We next present the state machine code generator in conjunction with a specific example, namely a self-serve gas station pump. The state machine code generator only requires the state names and the name of the state space (or application). The user interface is rather straightforward.

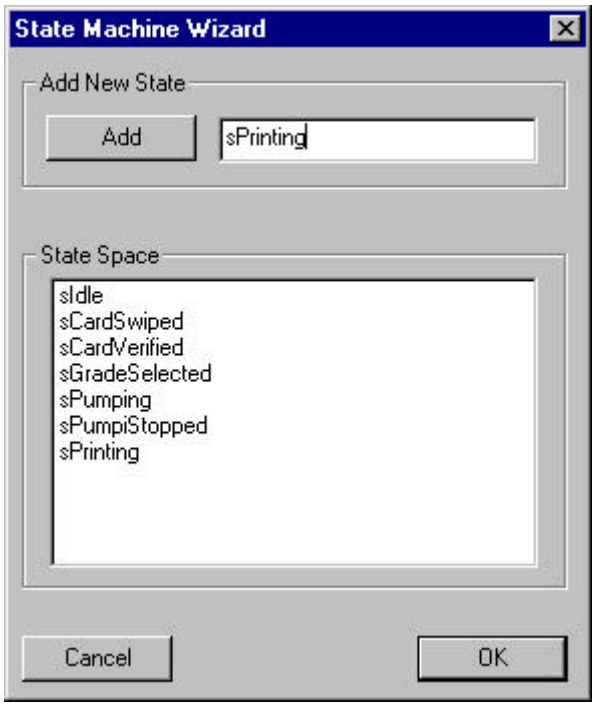

Figure 2. Inputting user-given state names to generate code for the state machine.

The user simply types state names into an edit field, which are then added to a list box. In case of an error, the states in the list box may be selected and deleted. Once all states names are entered, another dialog box asks for the state space name.

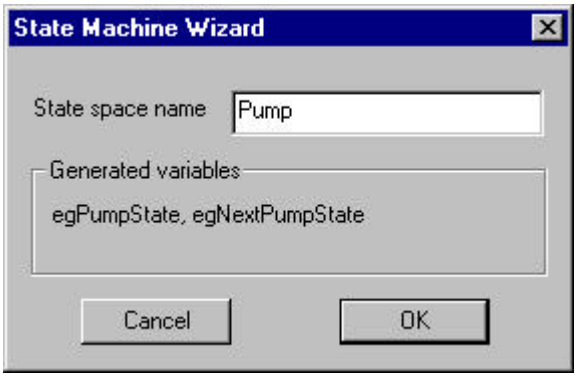

Figure 3. The name of the state space is used in defining global variables.

The state space name, here “Pump” is used in constructing the two global state variables as egPumpState and egNextPumpState. These variables and states are inserted into the corresponding fields of generated code.

The self-service gas pump follows a few simple operations. The user swipes a credit card, makes a selection, pumps the gas, and finally receives a receipt. Instructions are typically given on a simple character display, and a few pushbuttons are used to receive menu selections. The pump has other inputs, such as switches to sense the removal and insertion of the dispenser nozzle and

a fuel flow sensor, which we will omit for brevity. The following flowchart depicts a simplified set of operations. Note that the timeout feature is liberally used throughout the process.

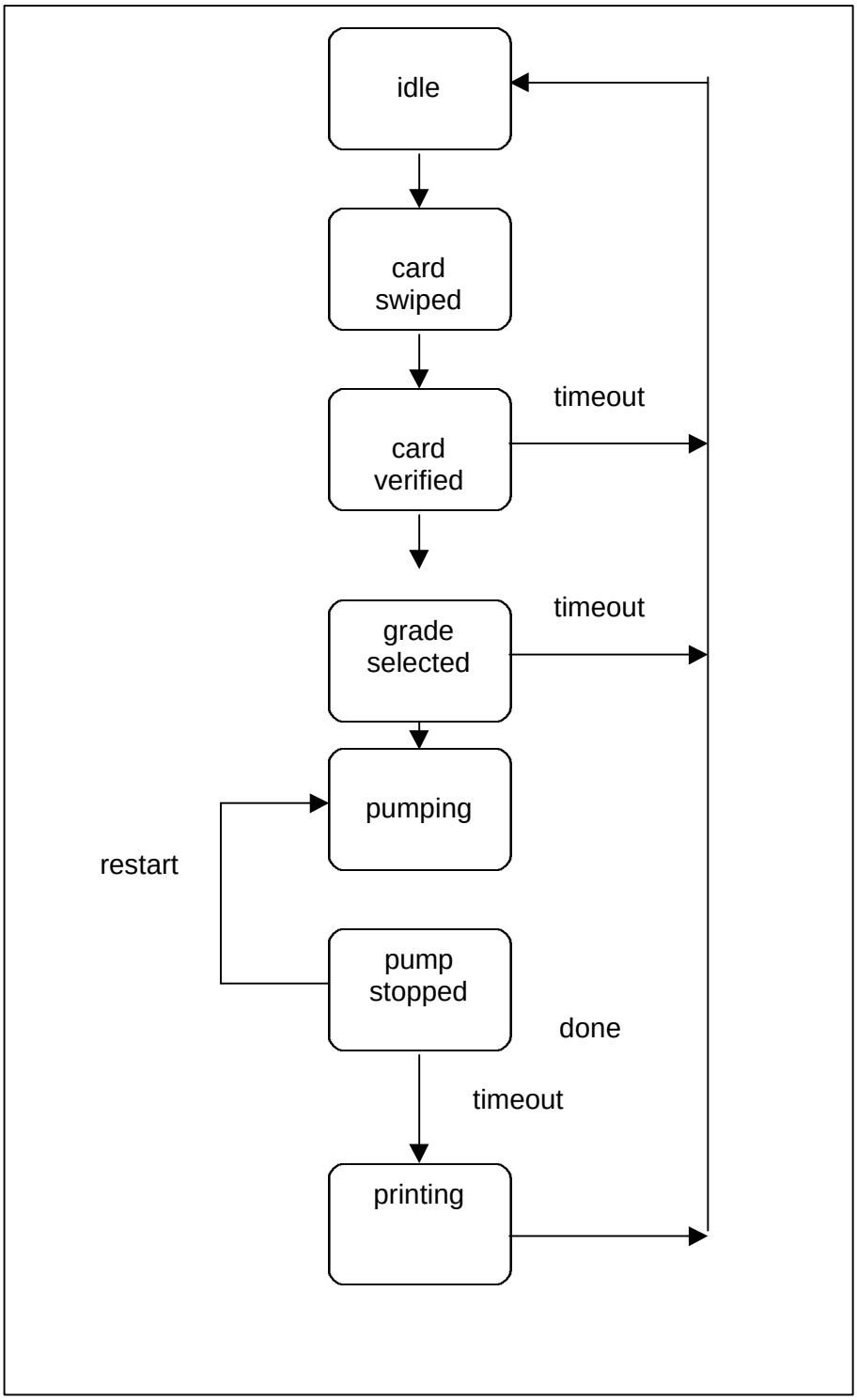

Figure 4. A simplified flowchart of self-serve gas pump operations.

The model has 7 states: *idle*, *card_swiped*, *card_verified*, *grade_selected*, *pumping*, *pump_stopped*, and *printing*. The process starts in the *idle* state. When the customer swipes a credit card the process enters state *card_swiped*. The pump then attempts to verify the card number and account. In this simplified version, we assume that the card is always verified and the process enters the state *card_verified*. In an actual application, if the card is not verified, an error message would be displayed and the process would return to *idle*. Once in the *card_verified* state, a query is displayed and the customer is expected to select the grade. If the selection is made in the allowed time the process enters state *grade_selected*. In this state, the customer is expected to start the pump within a given period. If the pump starts, the process enters state *pumping* and waits for the pump to stop. Note that we need to wait a little in state *pump_stopped* since the customer may not have completed the operation but have simply paused. If the pump restarts within a given grace period, we return to state *pumping*. Otherwise the process enters and remains in the state *printing* while a receipt is produced. Upon the print completion, the process returns to the idle state. Whenever a timeout occurs, the process returns to the idle state to repeat the cycle over. Note that this is a simplified set of operations. For example, the pump may accept debit cards, in which case, the customer needs to enter a personal identification number. Similarly, most pumps ask the customer if a receipt is desired. The timeout periods and timeout events play a fundamental role in this application. The polling approach is justified since the timeouts are in the order of tens of seconds.

The enumerated data type in C is a convenient way to define the states.

```
enum PUMPSTATE { sIdle,
                 sCardSwiped,
                 sCardVerified,
                 sGradeSelected,
                 sPumping,
                 sPumpStopped,
                 sPrinting
                 } egPumpState, egNextPumpState;
```

We define the states as well as two global variables to keep track of the current and next states.

We will take a top-down look at the code. Our main program is simply an endless loop that iterates through the state machine. Each iteration is accomplished by the routine is called StepStateMachine().

```
void StepStateMachine(void){

// get inputs -- this may change egNextState
 if(egPumpState==egNextPumpState) ScanEvents();

// execute state actions
 if(egNextPumpState!=egPumpState)
  {
   ExitState(egPumpState);       // leave current state
   EnterState(egNextPumpState);  // enter new state
   egPumpState=egNextPumpState;  // update state
   }
  else StayInState(egPumpState); // repeat last state
}
```

The actions to be taken at each iteration depend on the current and next states. Following our assumptions, the next state may change as a response to an external input, or internally by the state machine. We assume the transitions are instantaneous. Thus, the state may not change during the entry and exit activities.

Provided that the current state is the same as the next state, StepStateMachine() scans the inputs. The next state may change in ScanEvents(). If the current state is still the same as the next state the function StayInState() is called. Otherwise, the state machine undergoes a transition. The exit activity is performed in ExitState() with its parameter set to the current state. Next the entry activity is performed by EnterState() with its parameter set to the next state. Finally, the global variable denoting the current state is equated to that of the next state.

In the demo, we simulate the actual inputs by keyboard characters. ScanEvents() inspects the microcontroller's serial port. If a character is received PeekChar() returns a nonzero value (TRUE). In this case, the character is removed from the serial port buffer by GetChar(). The next state is determined as a function of the received character.

```
void ScanEvents(void){
// enter your code : scan inputs and set egNextPumpState...

char c;

// check user input at serial port
 if(!PeekChar()) return;
 c=GetChar();
 switch(c)
  {
   case 'c': if(egPumpState==sIdle) egNextPumpState=sCardSwiped;
             break;

   case '1':
   case '2':
   case '3': if(egPumpState==sCardVerified)
              egNextPumpState = sGradeSelected;
             break;

   case 'b': if(egPumpState==sGradeSelected)
              egNextPumpState = sPumping;
             if(egPumpState==sPumpStopped)
              egNextPumpState = sPumping;
             break;

   case 'e': if(egPumpState==sPumping)
              egNextPumpState = sPumpStopped;
             break;
   }
}
```

The three functions ExitState() and EnterState() and StayInState() have the same structure. The code generator writes this structure. These functions are all based on a single switch statement whose expression is the function parameter (the state). The switch statement with the specified states is automatically generated by the code generator. The programmer needs to enter application-specific code into each case action. The function EnterState(), including the application-specific code, is given below.

```
void EnterState(enum PUMPSTATE es){
// enter your code : call entry activity functions...
 switch(es)
  {
   case sIdle :
    SendStr("Idle state... press 'c' (card swipe) to begin cycle...\n");
    break;

   case sCardSwiped :
    SendStr("Card swiped... Waiting for verification...\n");
    StartSTimer(1,1000);   // verification in 1 second...
    break;

   case sCardVerified :
    SendStr("Card verified... choose grade ('1','2', or '3')...\n");
    StartSTimer(0,3000);   // timeout in 3 seconds...
    break;

   case sGradeSelected :
    SendStr("Grade selected... press 'b' to begin pumping...\n");
    StartSTimer(0,3000);   // timeout in 3 seconds...
    break;

   case sPumping :
    SendStr("Pumping... press 'e' to end pumping...\n");
    StartSTimer(0,10000);   // timeout in 10 seconds...
    break;

   case sPumpStopped :
    SendStr("Pump stopped... press 'b' to start pumping again...\n");
    StartSTimer(0,1000);   // grace period is 1 second...
    break;

   case sPrinting :
              SendStr("Pump stopped... Printing receipt...\n");
    StartSTimer(1,1000);   // printing takes 1 second...
    break;
   }
}
```

Upon entry to a state, we display a message and depending on the state, launch a software timer. The timer is used to keep track of timeouts. In two cases, namely in states *card_swiped* and *printing*, the timer is used to simulate the time it takes to authorize the card and to print the receipt. The ExitState() function is really not needed in this example. We only insert code into the action of case corresponding to state *sPrinting*. We simply display a message that the cycle is over.

Besides in the function ScanEvents(), the next state may change in the function StayInState(). Again, the switch statement is generated by the code generator. Generally speaking, we check for timeouts. No action is taken if no timeout has occurred. If there is a timeout, we abort the cycle and return to the idle state. The two exceptions, namely the states where the timer is used to simulate the card authorization and the printing of the receipt do not abort the process.

```
void StayInState(enum PUMPSTATE es){
 switch(es)
  {
   case sIdle :
    break;
```

```
    case sCardSwiped :
     if(EXPIRED(0))
      egNextPumpState=sCardVerified; // verification completed
     break;

    case sCardVerified :
     if(EXPIRED(1))
      {
       SendStr("Timeout !\n");
       egNextPumpState=sIdle; // timeout
       }
     break;

    case sGradeSelected :
     if(EXPIRED(0))
      {
       SendStr("Timeout !\n");
       egNextPumpState=sIdle; // timeout
       }
     break;

    case sPumping :
     if(EXPIRED(0))
      {
       SendStr("Timeout !\n");
       egNextPumpState=sIdle; // timeout
       }
      break;

    case sPumpStopped :
     if(EXPIRED(0)) egNextPumpState=sPrinting;
     break;

    case sPrinting :
     if(EXPIRED(1)) egNextPumpState=sIdle; // printing completed
     break;
    }
}
```

Our code generator creates the functions with the switch statements. The switch statements have a separate case clause for each of the states. The application-specific code is written mostly as case action statements. It is also recommended that entry and exit activities be written as separate functions for each state. This way, these functions may be called at the proper case actions.

It is noteworthy that the effort associated with writing a custom code generator turned out to be much less than expected at the outset. Granted, it is initially awkward to think about code generator code, which in turn will produce source code. Nonetheless, the entire process seems to be very orderly, especially if one starts with a sample output of the code generator. Writing the code generator to produce this sample output by a series of “printf” statements is almost trivial. The customization process may then be undertaken iteratively as you add features to your dialogs and incorporate the corresponding values in the “printf” statements.

References

1. Complete source code of this article and further information is available from Rigel Corporation, www.rigelcorp.com.
2. DAvE Home Page, http://www.smi.siemens.com/DAvE/newsDAvE.html

3. Information on Xilinx, Inc. CPLD and FPGA development tools are available at http://www.xilinx.com.
4. Information on BetterState is available from the Integrated Systems, Inc., website http://www.isi.com/products/betterstate.

Sencer Yeralan, P.E., Ph. D.

About the author

Dr. Yeralan has been teaching industrial automation and control at the University of Florida for the last ten years. He designs the C166 family evaluation and OEM boards made by Rigel Corporation. He may be reached at spy@rigelcorp.com.