Revision: 16 SEP 2015 (JFF)
Richard W. Wall, University of Idaho, rwall@uidaho.edu

www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 3: Software Based Finite State Machines

Purpose

The purpose of this project is to investigate the application of software based state machines to controlling the speed, direction of rotation, and operational mode of stepper motors. This project requires knowledge of concepts presented in Projects 1 and 2 to provide real-time open loop control.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB IDE
3. Understanding of Finite State Machines
4. IO pin control (see Project 1)
5. Time control of computer program execution (See Project 2)

Equipment List

1. Cerebot 32MX7cK processor board with USB cable
2. PmodSTEP
3. Stepper Motor (5V – 12V, 25Ω, unipolar or bi-polar)
4. Logic analyzer, or oscilloscope (Suggestion - Digilent Analog Discovery)

Software Resources

1. XC32 C/C++ Compiler Users Guide
2. PIC32 Peripheral Libraries for MBLAB C32 Compiler
3. Cerebot MX7cK Board Reference Manual
4. PIC32 Family Reference Manual Section 14: Timers
5. MPLAB ® X Integrated Development Environment (IDE)\
6. C Programming Reference

References:

1. Microchip AN907 – Stepper Motor Fundamentals.
   http://ww1.microchip.com/downloads/en/AppNotes/00907a.pdf

2. Introduction to Stepper Motors:
   http://www.omega.com/auto/pdf/REF_IntroStepMotors.pdf

3. **"Control of Stepping Motors A Tutorial",** Douglas W. Jones**,**
   http://www.cs.uiowa.edu/~jones/step/index.html

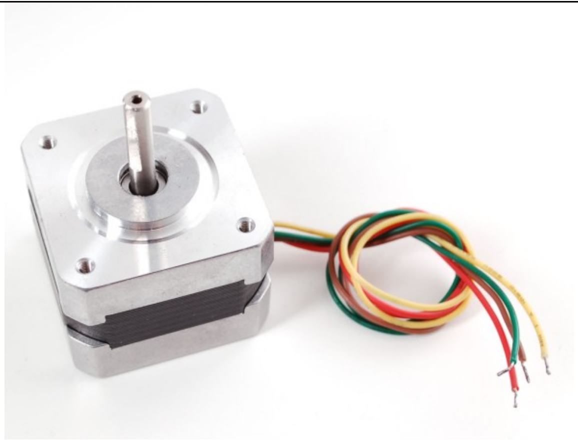
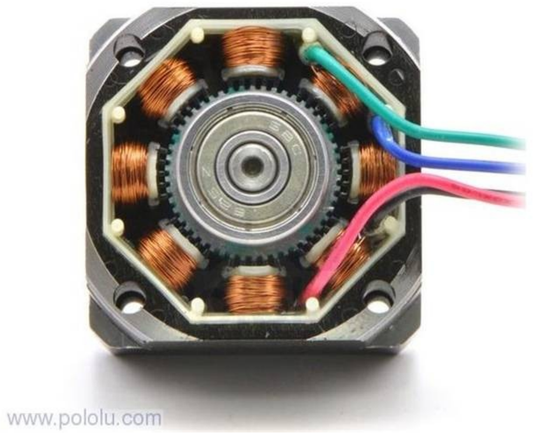
4. Stepper Motor Theory of Operation, National Instruments,
   http://zone.ni.com/devzone/cda/ph/p/id/248

5. **Stepping Motor Types, Douglas W. Jones,**
   http://www.cs.uiowa.edu/~jones/step/types.html

Introduction to Stepper Motors and Finite State Machines

After a search of the internet, one will find that the stepper motor is one of the most often used examples of an application of a finite state machine (FSM). Applications that use stepper motors include robotics, disk drives, and office products such as laser printers and copiers.

Stepper motors, like the one shown in Fig. 1, are variable reluctance electric motors that are designed to control angular position of the rotor shaft in discrete steps. The stepper motor consists of two sets of field windings positioned around a permanent magnet rotor as illustrated in Fig. 2. The combinations of voltage applied to the four control terminals of the field windings control the magnitude and direction of the current through the windings as illustrated in Fig. 3. The current through the windings create an electromagnet. The motor shaft rotates to a position that minimizes the reluctance path between the field winding electromagnet north and south poles.

 Fig. 1. Photograph of typical stepper motor	 Fig. 2. Photograph field coils and rotor of a stepper motor

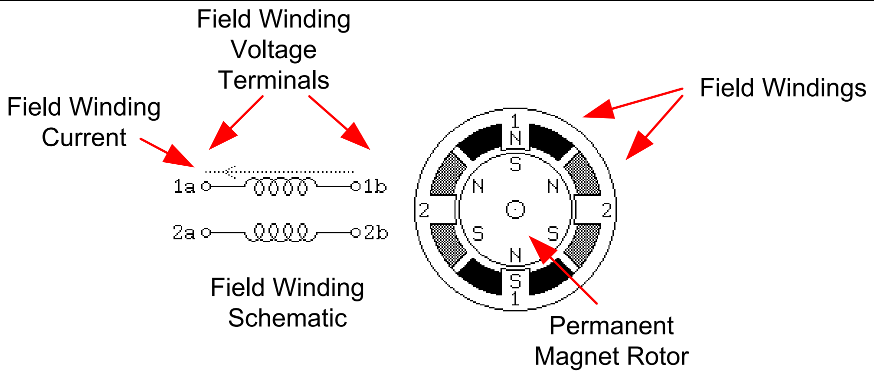


Fig. 3. Bipolar Stepper motor diagram

Considering the combinations of voltages on the winding terminals as possible control states, there are only eight states that produce current in the field windings as shown in Table 1 below. In order to move the rotator shaft from one stable position to the physically adjacent stable position, the control voltages must switch to one of four out of the eight possible combinations of voltages. The action of moving from one stable position to an adjacent stable position is referred to as either a full-step or a half-step. Half-step increments are half the angular rotation of full-steps. Repeating a sequence of full or half step movements at a uniform rate will cause the rotator shaft to rotate at a constant speed in discrete steps.

Table 1. Stepper motor control codes

Step Control		Winding Voltage			
Step Name	Hex Code	“1a”	“1b”	“2a”	“2b”
S0_5	0x0A	H	L	H	L
S1	0x08	H	L	L	L
S1_5	0x09	H	L	L	H
S2	0x01	L	L	L	H
S2_5	0x05	L	H	L	H
S3	0x04	L	H	L	L
S3_5	0x06	L	H	H	L
S4	0x02	L	L	H	L

The stepper motors used in this project are designed to require 100 full steps for the rotor shaft to complete one full revolution or 3.6 degrees per step. 200 half-steps are required to make one revolution or 1.8 degrees per half-step. The first column in Table 1 is a label assigned to the state. The second column is the hexadecimal code that will set the processor IO pins to control the voltages on the terminals of the windings. The last four columns in Table 1 represent the combinations of voltages on the field windings that produce stable rotator shaft positions. The letter “H” denotes a high voltage and the letter “L” denotes a low voltage. As shown in Fig. 3, current flows through a motor coil when there is a voltage difference across the winding. The voltage combinations for step 3 in Table 1 represent the combination to produce the current flow shown in Fig. 3.

Binary codes are assigned such that we replace the letter “H” with a binary 1 and the letter “L” with a binary 0. The values shown in column 2 of Table 1 are the hexadecimal equivalent of the binary representation of the field winding voltages. The four winding designations shown in Table 1 are assigned to IO pins as shown in Fig. 5 found in Appendix A. The stepper motor will move to the nearest stable position generated by the voltages associated with hexadecimal codes shown in the second column. The stepper motor will be held in a fixed position until the voltages on the windings change.

The PmodSTEP driver module was designed to use IO PORT B pins 7 through 10 that are assigned designations SM1 through SM4 to control the voltages on the four stepper motor field windings. (See Appendix A for PmodSTEP connection details.) The hexadecimal codes listed in Table 1 must be shifted left such that the bit representing winding “2b” appears on Port B pin 7, the bit representing winding “2a” appears on Port B pin 8, the bit representing winding “1b” appears on Port B pin 9, and the bit representing winding “1a” appears on Port B pin 10

If the motor is to rotate the in a clockwise (CW) direction using the full-step mode, the sequence of output codes that must be sent to the motor are represented by steps S1, S2, S3, S4, S1, …. A 100 step per revolution motor will require the sequence of the four output combinations, S1 through S4, to be repeated 25 times to cause the rotator shaft to make exactly one revolution. If operating in half-step mode, then the eight step sequence of S0_5, S1, S1_5, S2, S2_5, … , S4 must be repeated 25 times for the rotator shaft to make a complete revolution. Sequencing through the output code the Table 1 in one direction (up or down) causes the rotator shaft to rotate in one direction. Reversing this sequence causes the rotator shaft to reverse direction.

Programming Concepts

In a C-based program, the '*main*' function consists of an infinite software loop that repeatedly executes a sequence of tasks inside the *while(1)* loop. A set of tasks must be repeatedly completed in the prescribed order within the "while(1)" loop to correctly control the stepper motor operation. The actual steps required and the order in which they are executed are prescribed by the application requirements. One possible sequence of operations to control the stepper motor is shown in Listing 1.

Listing 1. Five operations used to control the stepper motor.

1. Sense the button status of the control by polling the inputs that are connected to the buttons
2. Map the button status to specific direction of rotation and step mode
3. Determine the new stepper motor control output
4. Output code to the stepper motor
5. Delay the correct amount of time using one of the software delay approaches developed in Project 2.

In this project, we are implementing a simple form of real-time open loop control. The term "real-time" implies that all tasks must be completed in a specified time. The processor completes the code for the first four of the five operations listed in Listing 1 as quickly as the processor can execute the code. The fourth operation causes the stepper motor to physically move. The fifth causes the processor to wait before repeating the first four steps again. Note that for the control implementation described in Listing 1 the buttons are sampled at the step rate. Hence, the slower the motor is moving the less frequently the buttons are polled. One may consider ways to modify the tasks listed in Listing 1 to make the control more responsive.

The precision of the stepping speed depends on the accuracy of the software delay loop and the amount of time required for processing the steps 1 through 4. One of the weaknesses of using software delays in control loops is the variance in loop speed due to changing execution times for the other tasks in the infinite task processing loop. This is also true of the hardware assisted delay loops.

The time delay function using the hardware assisted approach employs polling of the core timer. Because the processor is always polling inputs and timers or just executing a "*for"* loop that does nothing else except to consume time, the processor has no time to do anything else. This limitation will be rectified by changes to our methods of detecting events in Project 5.

There are three parameters that will be used to control the stepper motor: the step direction as either CW or CCW, the step mode as either FULL or HALF, and the step delay that determines the rotator shaft rotational speed. For this project, we will be using a fixed speed of 15 RPM.

The next output code that is needed to move the motor to the next position depends on the present rotor position. So the present position must be remembered from step to step. The need to 'remember' the step code that set the present rotator shaft position implies memory and the concept of a "state". Hence a state machine will be used to determine the code to move rotator shaft to the next position. The output code, corresponding to the next state, depends upon the present state and the button inputs. The frequency of transitions is controlled by the delay function, similar to a clock period in a hardware based FSM. Additional information regarding software implementation of finite state machines is provided in Appendix B.

The stepper motor driver module, PmodSTEP, shown in Appendix A has eight test points that are connected to the eight LEDs labeled LEDA through LEDH. The physical location of the test points is shown in Fig. 4. The wiring diagram provided by Fig. 5 shows that the pins that control LEDE through LEDH are also connected to a driver IC that amplifies the voltage and current switching capability needed to drive the stepper motor. The CerebotMX7cK.h header file defines the pin constants for LEDA through LEDH and also SM1 through SM4 as shown in Table 2 of Appendix A. These assignments assume that the PmodSTEP is connected to the Cerebot MX7cK Pmod jack JA.

Instrumentation for monitoring the period between steps (ms/step) is provided by toggling LEDB on the PmodSTEP module each time a new step is generated. This bit can be toggled using the LATBINV instruction. However, care must be taken when changing the PORT B outputs using an assignment to LATB to change the stepper motor shaft position. You cannot simply write the stepper motor control code to Port B using the instruction, "PORTB = code;" Doing so will also clear and out PORT B bits used for instrumentation.

Since LEDA, LEDB and SM1 through SM4 outputs all share the same IO port, the bit state for LEDA and LEDB must be preserved when setting SM1 through SM4. This requires a read–modify-write sequence in software. The bit set and bit clear operations should not be used because the two operations result in two different outputs being generated instead of just one. One possible method for implementing a read-modify-write sequence of code that was presented in Appendix A of Project 1.

Project Design

You will write a program to control the direction and step-mode of a stepper motor at the fixed speed of 15 RPM as per the rules shown in Table 2. Your program should contain the seven software functions whose functionality is described in Listing 2. Functions numbered 3 through 7 will be continuously repeated, in order, within a *while(1)* software control loop. Note: Having completed Projects 1 and 2, you have already written software that partially implements all the functions in this list except function 5. The state diagram shown in Appendix B is to assist you in developing the stepper motor control function that implements the FSM.

Table 2. Stepper motor control table

Inputs		Control Modes	
BTN2	BTN1	DIRECTION	STEP MODE
Off	Off	CW	FS
Off	On	CW	HS
On	Off	CCW	HS
On	On	CCW	FS

Listing 2. Task list for Project 3. Create one function for each task.

1. "main" – the task control function responsible for calling the support functions listed below
2. "system_init" – initializes all needed resources
3. "read_buttons" – reads the status of BTN1 and BTN2
4. "decode_buttons" – determines the values of stepper_direction, stepper_mode, and stepper_delay using the rules specified in Table 2
5. "stepper_state_machine" – determines the new output code for the stepper motor
6. "output_to_stepper_motor" – sends the four bit code to the stepper motor IO pins
7. Implement a ms software delay that toggles LEDA each ms
8. "sw_delay" – delays the number of milliseconds to generate a 15 RPM rotational speed and toggles LEDB each time a step is taken

Project Testing

The speed specifications are given in RPM must be to a delay with units of milliseconds per step. The following equation (Eq. 1) provides the needed conversion formula for a speed that is specified as X RPM. The factor, "MODE" in Eq. 1 is either 1 for full-step mode or 2 for half-step mode.

T_{DELAY} (ms/step) = 60000 ms/min / (X rev/min * 100 steps/rev * MODE) Eq. 1

The speed is specified as a constant 15 RPM for this project but the mode parameter is set according to the conditions provided in Table 2. Hence T_{DELAY} will need to be adjusted to keep the motor speed constant.

For testing the direction control, one simply needs to observe the shaft of the stepper motor to verify that the direction changes when BTN2 is pressed. Use Table 3 of the project report to record your results.

Table 3. Stepper motor delay per step verification table.

Inputs		**Control Modes**		
BTN2	BTN1	STEP MODE	STEP DELAY Calculated	STEP DELAY Measured
Off	Off	FS	40ms	
Off	On	HS	20ms	
On	Off	HS	20ms	
On	On	FS	40ms	

Appendix A: PmodSTEP

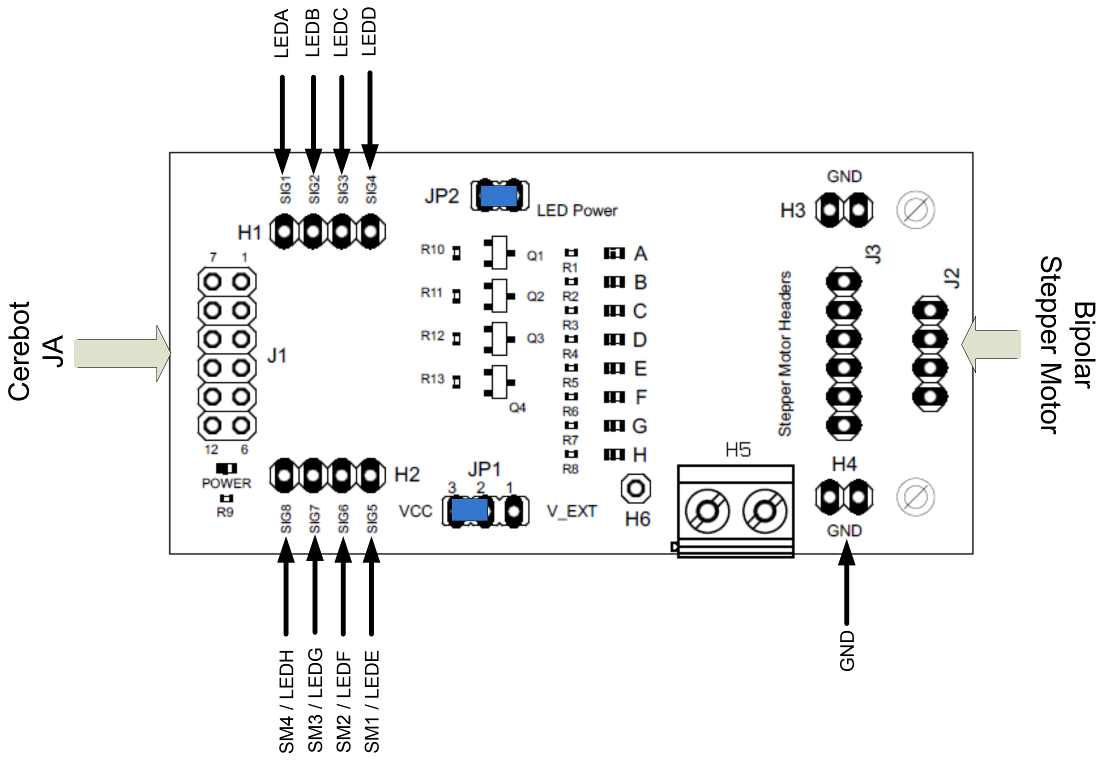


Fig. 4. PmodSTEP parts layout

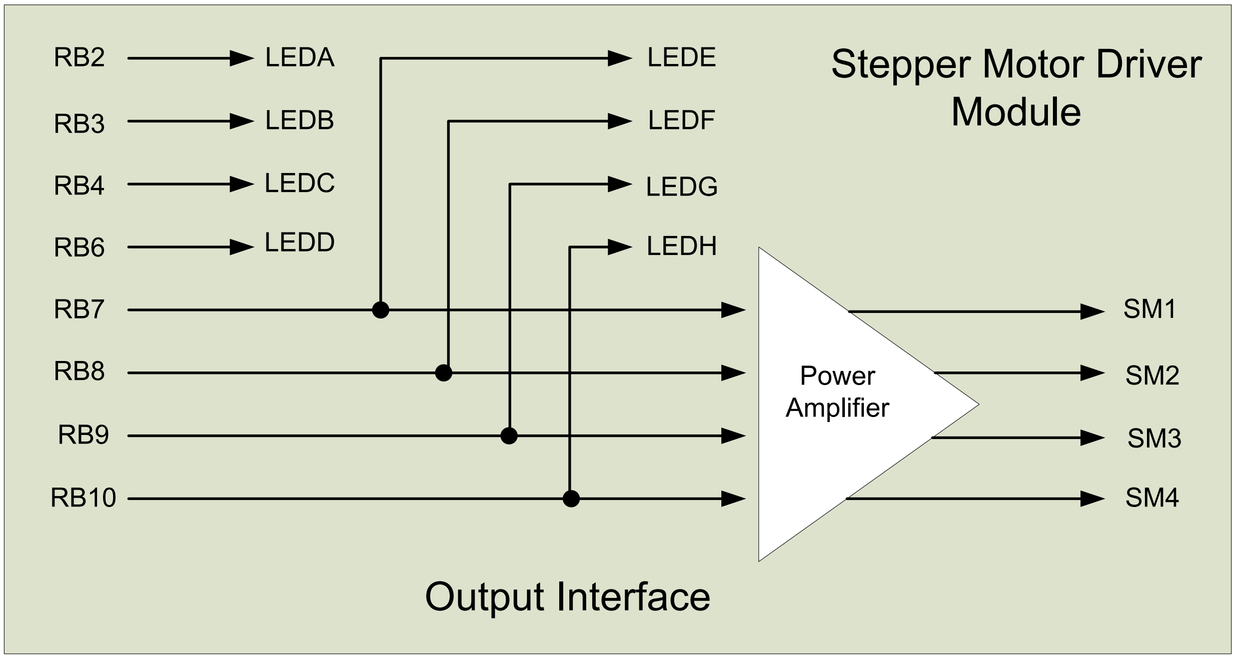


Fig. 5. PmodSTEP wiring diagram

Table 2. LED – Pmod Port JA Pin and PIC32MX7 IO Port B Assignments

LED	JA Pin #	PIC32MX7 Port B Pin #
LED A	1	RB2
LED B	2	RB3
LED C	3	RB4
LED D	4	RB6
LED E / SM1	7	RB7 (winding 2b)
LED F / SM2	8	RB8 (winding 2a)
LED G / SM3	9	RB9 (winding 1b)
LED H / SM4	10	RB10 (winding 1a)

Appendix B State Transition Diagram Model for Stepper Motor Control

The state transition diagram shown in Fig. 6 graphically represents the states and transition conditions for the stepper motor control. States represented by S1 through S4 are "full-step states". States represented by S0_5 through S3_5 are "half-step" states. Each state outputs a unique combination of four bits to the stepper motor as specified by Table 1**.**

The transitions that are strictly between full-step states and strictly between half-step states cause the stepper motor to rotate one full step. For example, a transition from state S1 to S2 causes the motor to rotate a full step. Likewise a transition between S1_5 and S0_5 also causes the motor to move a full step rotation. A transition between half-step and full-step states cause a half-step angular rotation. Such is the case for a transition from state S3_5 to S3.

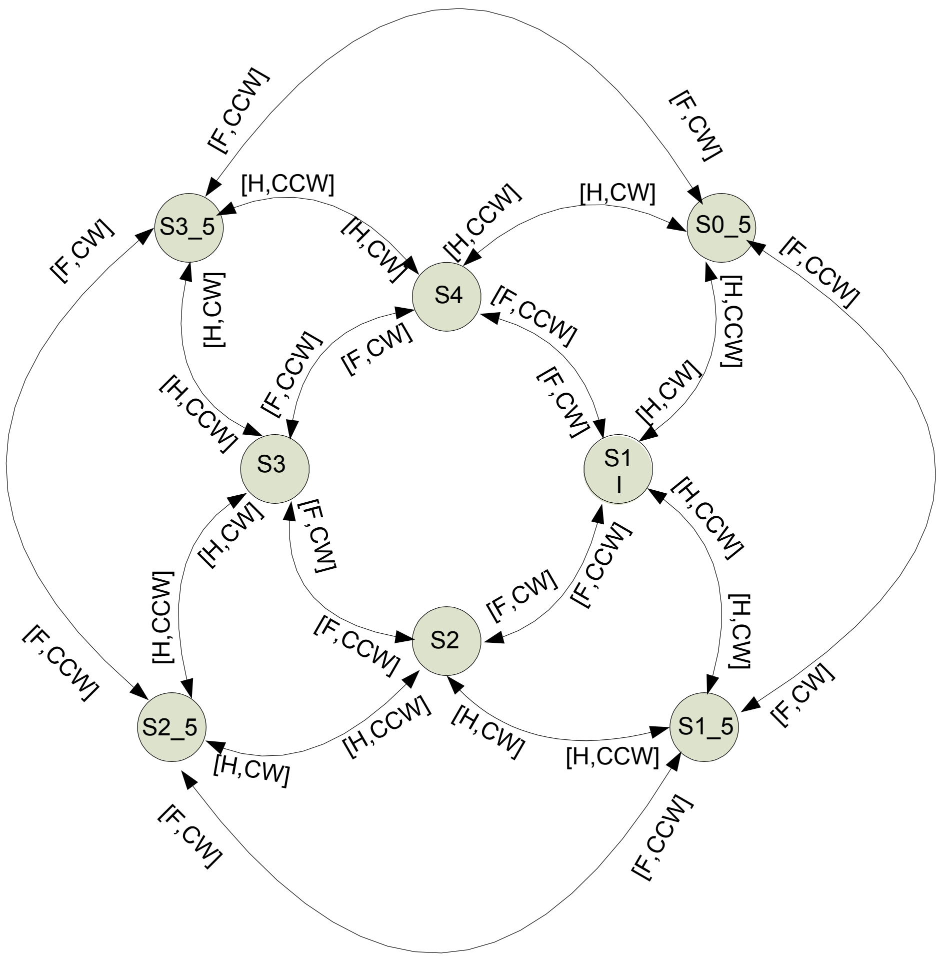


Fig. 6. FSM diagram for stepper motor step and half-step control. Each state is assigned an on-entry output defined by Table 1

A transition is generated each time the FSM function is called. Each transition is identified by two parameters inside square brackets that are use to determine the step direction and mode. Only transitions shown on the state diagram will result in the stepper motor rotating in a predictable manner. Each state has four possible transitions to a new state. The transition path to the new state is controlled by the clockwise (CW) and counter clockwise (CCW) direction controls as well as the full (F) or half (H) step mode. Whenever a new state is entered, it generates an output code, shown in Table 1.

To illustrate how state diagrams can be used to assist in the code development, consider the state diagram described by Fig. 7 that models using a momentary contact push button to implement a push-on / push off switch. There are two possible output conditions: the switch is either closed (1) or open (0). The condition of the momentary contact push button input is either pressed (1) or not pressed (0). This design requires four states labeled SA through SD. The initial state of the switch is identified by the transition from state "I". Each state identified by a circle has a name and an output condition shows as a value below the state name. For example, the initial state, SA, has an output condition 0. "B" represents the stable button condition (all button contact bounce has been eliminated.) The state diagram shows that the switch output changes whenever the button makes a transition from the not pressed to the pressed condition.

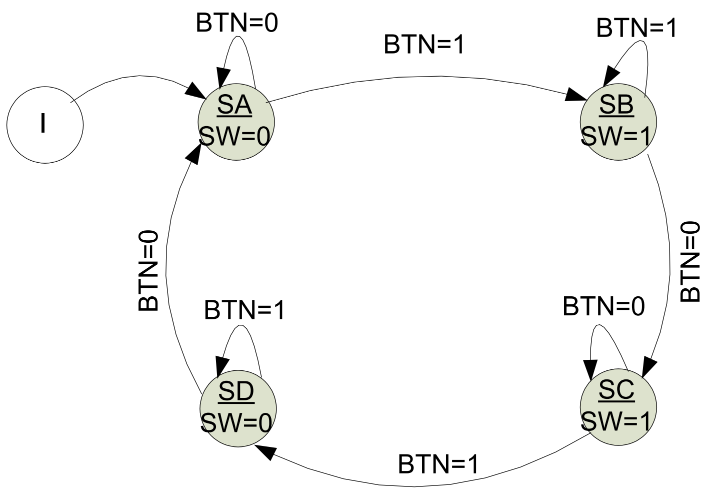


Fig. 7. Push-on / push-off button state diagram

Listing 1 is the C code that implements the state diagram shown above using the "switch-case" program control structure. Line 13 defines the states using a enumeration declaration. The state variable, *pb_state*, and the switch output variable, *sw*, are both declares as static variables that retain the previously assigned value between successive calls to the *pb_sm* function. The variable, *pb_state*, provides the necessary state memory. The switch output variable, *sw,* is declared static for more efficient program execution by eliminating the need to set the variable each time the function is called.

The cases used in the switch-case control structure shown in Listing 1 have a one to one correlation with the states defined in Fig. 7. Each time the function is called, a transition will be taken to a next state. The next state transition is either to a different state or back to the current state. The conditional *if* statements in lines 19, 26, 32, and 40 control the transition to a different state. If no transitions conditions are met, the state remains unchanged. This represents the transition to the same state for the next state condition.

Listing 1. C code for implementation of the push-on / push-off button design.

```
/* pb_sm Function Description ***********************************************
 * KEYWORDS:        push button, FSM, state machine
 * DESCRIPTION:     Implements a push-on / push-off switch action using a
 *                  simple momentary bounceless push button.
 * PARAMETER1:      int btn - button condition: 1 = pressed, 0 = relaxed
 * RETURN VALUE:    switch on/off output: 0 = open, 1 = closed.
 * NOTES:           This state implementation generates output on state exit
 *                  transitions.
 *
 * END DESCRIPTION **********************************************************/
int pb_sm(int btn)
{
    enum {SA=0, SB, SC, SD};     /* Declaration of states */
    static int pb_state = SA;    /* Push button state variable initialized to SA */
    int sw = 0;                  /* Switch output initialized to open */
    switch(pb_state)
    {
        case SA:
            if(btn)              /*Is button pressed? */
            {
                pb_state = SB;  /* Transition to next state */
                sw = 1;         /* Output change to closed (1) on state exit */
            }
            break;
        case SB:
            if(!btn)            /*Is button not pressed? */
            {
                pb_state = SC;  /* Transition to next state */
            }
            break;
        case SC:
            if(btn)              /*Is button pressed? */
            {
                pb_state = SD;  /* Transition to next state */
                sw = 0;         /* Output change to open (0) on state exit */
            }
            break;
        case SD:
            if(!btn)            /*Is button not pressed? */
            {
                pb_state = SA;  /* Transition to next state */
            }
            break;
    }
    return sw;                  /* Return switch output */
} /* End of pb_sm */
```