PLECS

Tutorial

Efficient PWM Generation

Modulator implementation using the C-Script block and a state machine

Last updated in tutorials release 1.0

1 Introduction

The C-Script block is not only useful for implementing mathematical and control functions, but also for implementing state machine programs. State machine programs are useful for generating switching patterns and for sequencing controller modes. In this exercise, you will implement a state machine program with the C-Script block that creates a symmetrical pulse width modulation (PWM) signal with a blanking delay between switching transitions.

Using a state machine program with a variable time step setting to generate PWM is very efficient since the PWM state machine program is only called during a switching transition. This technique is more efficient than the standard approach of continuously comparing the modulation index with a triangular carrier wave. With a comparator-based approach, intermediate simulations steps must be taken around each transition point in order for the solver to determine the exact transition time of the comparator output.

Before you begin Ensure the file `test_inverter.plecs` is located in your working directory. You should also have the reference files that you can compare with your own models at each stage of the exercise.

2 Background

PWM is commonly used for controlling inverters such as the full-bridge single-phase inverter shown in Fig. 1. To generate the PWM waveform, a sinusoidal control signal known as the modulation index, m, is typically compared with a triangular PWM carrier signal as shown in Fig. 2. Different PWM switching strategies and switching signals are possible, but for this exercise, the PWM output is a switching signal in the set of $[-1, 0, 1]$ that controls all four switches in the inverter. A 1 turns on the switch pair S1 and S4 and a -1 turns on switch pair S2 and S3. With an output of 0, all switches are off. This is a bipolar switching strategy since the resultant output voltage is bipolar, having a value of V_{dc} or $-V_{\mathrm{dc}}$.

In a practical inverter, dead time is needed between switching transitions to ensure that physical delays during the turn on and turn off of the switching elements do not overlap and cause short circuits across the DC bus. Dead time control is achieved by waiting a finite time for one switch in the inverter leg to turn off before the other switch in the leg is turned on.

A PWM modulator with dead time can be modeled efficiently in PLECS using a state machine that is only called at switching transitions. For an ideal PWM modulator, only two switching transitions per cycle are needed, and with dead time, only two additional transitions are required.

3 Exercise: Ideal PWM

In this first exercise you will implement a state machine program that generates an ideal symmetrical PWM signal as shown in Fig. 3. For use in bipolar switching applications, the input signal is in the range $[-1,\ 1]$ and the output in the set $[-1,\ 1]$.

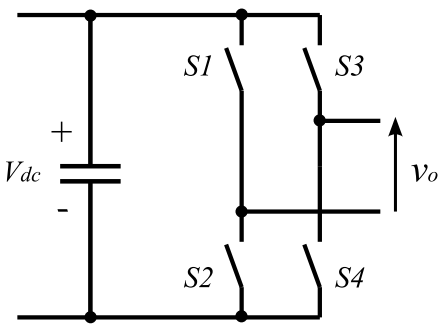

Figure 1: Full-bridge single-phase inverter.

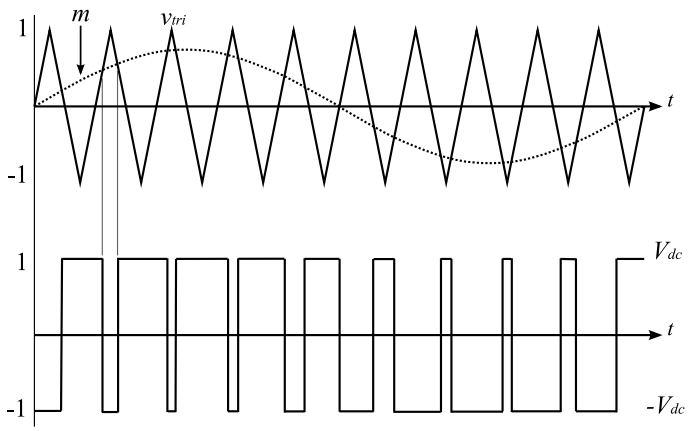

Figure 2: Generation of switching signal for bipolar sinusoidal PWM modulation. The PWM output controls the full-bridge switch pairs to produce a bipolar voltage output.

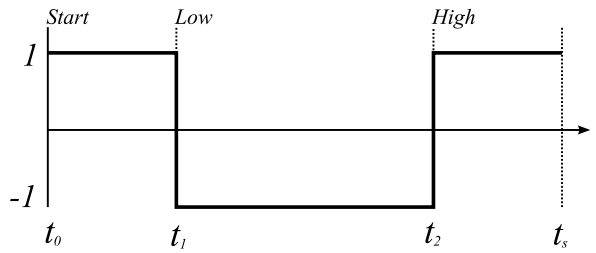

Figure 3: Timing of a symmetrical PWM cycle.

3.1 Operation of PWM state machine

The state machine program is summarized in the state diagram shown in Fig. 4. The state diagram comprises three states, **Start**, *Low* and *High*, where each state corresponds to a time instant at which the program is called.

The PWM state machine uses a discrete-variable hybrid time setting. The fixed time step setting creates regular calls, or hits, to the C-Script block at intervals of $1/f_s$. This setting is used to begin the switching cycle. The variable time step setting is used to set the transition times into the *Low* and *High* states.

To begin with, the fixed time step hit causes the state machine to enter the *Start* state. In the *Start* state, if the modulation index, m, is not equal to 1 or 0, the scaled modulation indices, m_1 and m_2, are calculated. The scaled modulation indices are transformed from the range $[-1,\ 1]$ into the range $[0,\ 1]$ using:

$$m_1 = (m+1)/2 \tag{1}$$

$$m_2 = 1 - m_1 \tag{2}$$

If m_1 or m_2 are less than or equal to 0, the PWM output is set and the state machine program loops back to the *Start* state. If m_1 and m_2 are within limits, the variable-step hit times, t_1 and t_2 are calculated as follows:

$$t_1 = t_0 + \frac{m_1}{2} t_s \tag{3}$$

$$t_2 = t_1 + m_2 t_s \tag{4}$$

The derivation of the hit time calculations for symmetrical PWM is graphically depicted in Fig. 7 of Appendix A.

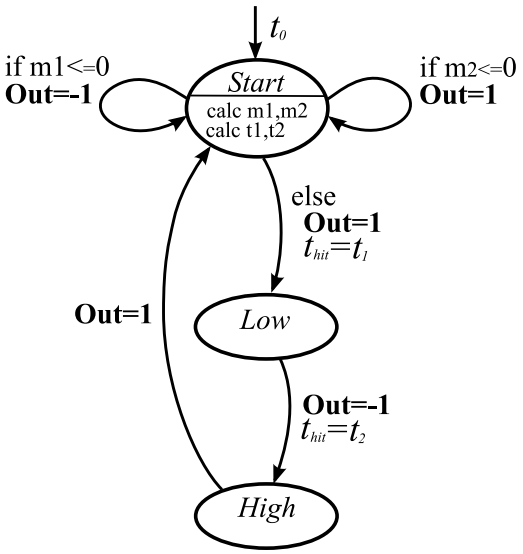

Figure 4: State diagram for symmetrical PWM sequence generation. The variable transition times into the *Low* and *High* states are manually defined by setting t_{hit}. At $t = 0$ or in the *High* state, the automatic fixed-step hit initiates the transition to the *Start* state.

3.2 Implementation

Your Task:

1 The C-Script block should be placed inside a Subsystem block to allow you to define a subsystem parameter, labeled **Switching frequency** and with the variable f_s, to set the PWM switching frequency. You will also need to pass $1/f_s$ to the C-Script block as a custom parameter.

2 The configuration settings for the C-Script block are shown in Fig. 5. Note that the **Sample time** parameter is configured with a discrete-variable hybrid time setting. The fixed time setting causes hit times at regular spaced intervals of $1/f_s$ and the variable time setting allows you to define the hit times for the PWM waveform transitions internally during each cycle.

3 To complete the configuration of the C-Script block, in the **Code Declarations** function map the input to `M`, the output to `OUT` and define a variable `TS` of type double. In the **Start function** assign the user parameter $1/f_s$ to `TS` using the macro `ParamRealData`.

4 To implement the state machine in C code, you should first define a data type named `state_type` and the possible names for the data type. This is done by creating an enumeration in the **Code Declarations** function as follows:

```
typedef enum {
  Start,
  Low,
  High,
}state_type;
```

5 You can now create variables of type `state_type` as you would with any other type, such as `int` or `double`. You should create a variable of this type named `NEXT_STATE` to keep track of the next state. Note that any variable defined in the **Code Declarations** function is global and therefore persistent.

6 Also in the **Code Declarations** function, include the header file *float.h*, which defines the largest theoretical machine representable number `DBL_MAX`, and assign this to a macro named `NEVER`. Also

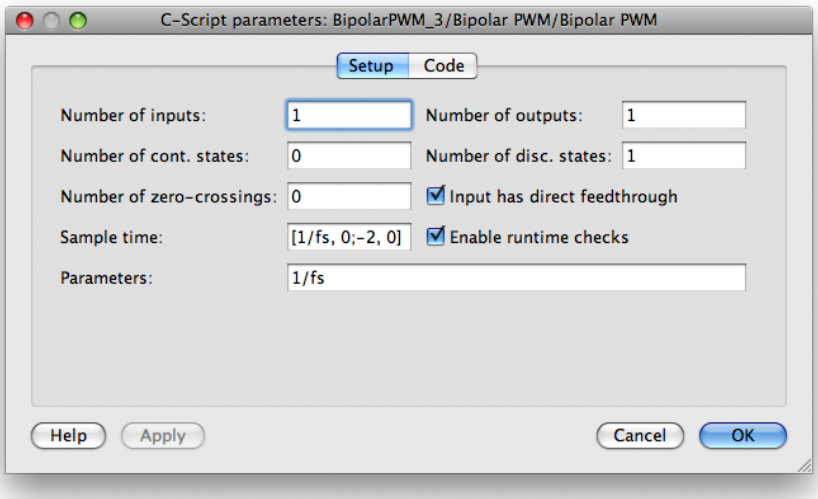

Figure 5: C-Script block configuration settings.

create a macro named `DMIN` to represent the smallest permissible duty cycle, and assign this to an arbitrarily small value of $1e{-}6$ s.

7 In the **Code Declarations** function, you should also map a variable named `STATE` to `DiscState(0)`. Note that the variable `STATE` is represented internally as a double, but can be cast as a `state_type` variable in order to allow a `STATE` variable to be compared with the state names rather than doubles.

8 In the **Start** function, set `STATE` to be in the *Start* state and initialize the internal macro `NextSampleHit` to `NEVER`.

9 In the **Output** function, define m_1 and m_2 as double precision numbers. Also define the variables t_1 and t_2 as static double precision numbers. The static keyword is needed as these variables must retain their values between **Output** function calls.

10 Implement the state machine logic in the **Output** function using the switch statement shown below. In each state, the PWM output and the `NextSampleHit` macro should be set and the `NEXT_STATE` defined. In the *High* state, set the `NextSampleHit` macro to `NEVER` to ensure that the next call to the state machine program is from the fixed-step setting.

```
switch ((state_type) STATE) {
  case Start:
  //Calculate m1, m2. Determine NEXT_STATE.
  //Calculate t1, t2. Set PWM output.
  //Set NextSampleHit
  break;

  case Low:
  break;

  case High:
  break;
}
```

11 You will need to implement some state transition logic in the *Start* state. If m_1 or m_2 are less than `DMIN`, set the PWM output to 1 or -1 and loop back to the *Start* state.

12 Don't forget to update `STATE` to `NEXT_STATE` in the **Update** function.

13 To test the PWM modulator, apply a $50\,\mathrm{Hz}$ sine wave with an amplitude of 1 to the input, and set the switching frequency to a low value of $1000\,\mathrm{Hz}$. You can set the simulation stop time to $20\ \mathrm{ms}$ and leave all other simulation parameters as their defaults. Compare the PWM output with Fig. 2.

At this stage, your model should be the same as the reference model, `cscript_modulator_1.plecs`.

4 Exercise: PWM with Blanking Time

In this exercise you will extend the PWM state machine from the previous exercise to include a blanking time, t_d, between the switching transitions. You will also limit the minimum high or low time of the PWM waveform to $d_{\min}$. You will need to modify the state diagram to add blanking states, *Blank1* and *Blank2* before states *Low* and *High*, respectively. The hit times for the state transitions are shown in Fig. 6.

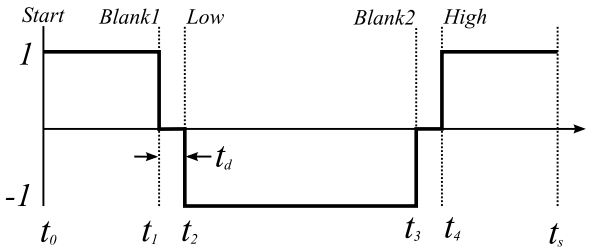

Figure 6: Timing of a single switching cycle with blanking time t_d between switching transitions.

Your task

1 Create new mask parameters in the subsystem labeled **Dead time ratio** (`dr`), and `Minimum duty cycle` ($d_{\min}$), and pass these to the C-Script block as custom parameters. Define variables of type `double` named `DEADTIME` and `DMIN` in the **Code Declarations** and assign the user parameter dr and $d_{\min}$ using the `ParamRealData` macro. Note that the custom parameter $1/f_s$ is entered as before and mapped to `TS`.

2 The new on and off ratios are adjusted for dead time by subtracting the dead time ratio, dr, from the values calculated in Eq. (1) and (2):

$$m_1' = m_1 - dr \tag{5}$$
$$m_2' = m_2 - dr \tag{6}$$

3 In the event that $m_{1,2} < d$, the calculated value for $m_{1,2}'$ will be negative and the hit time calculation will be incorrect. Therefore before you calculate the hit times, impose a lower limit on m_1' and m_2' of $d_{\min}$ and an upper limit of $1 - 2dr - d_{\min}$.

4 The hit times for each state transition, graphically derived in Appendix A, are calculated using:

$$t_1 = t_0 + \frac{m_1'}{2} t_s \tag{7}$$
$$t_2 = t_1 + dr\, t_s \tag{8}$$
$$t_3 = t_2 + m_2'\, t_s \tag{9}$$
$$t_4 = t_3 + dr\, t_s \tag{10}$$

5 Modify the state machine program to include the states *Blank1* and *Blank2*. When adding these states to the state machine model, you should also remove the two conditional loops from the *Start* state. These are no longer needed since the duty cycles are limited above zero.

6 Set the minimum duty cycle to 0.02, the dead time ratio to 0.01 and compare the functionality of your modulator with the previous version. Place the modulator in the test model `test_inverter.plecs`. Set the switching frequency to $25e3$ Hz and observe the inverter output voltage and load voltage. Is there any visible change in the load voltage as you increase the dead time ratio?

At this stage, your model should be the same as the reference model, `cscript_modulator_2.plecs`.

5 Conclusion

State machine programs are useful for creating output sequences such as a PWM switching signal with blanking time. In this exercise you learned how to generate a PWM signal with a state machine program that is executed only at the PWM transitions. This implementation is more efficient that an implementation based on a counter and compare program. The state machine concept is not only useful for pattern generation but can easily be adapted to respond to external rather than internal events for applications such as control system sequencing.

A Appendix: Derivation of Hit Times for Sampled PWM

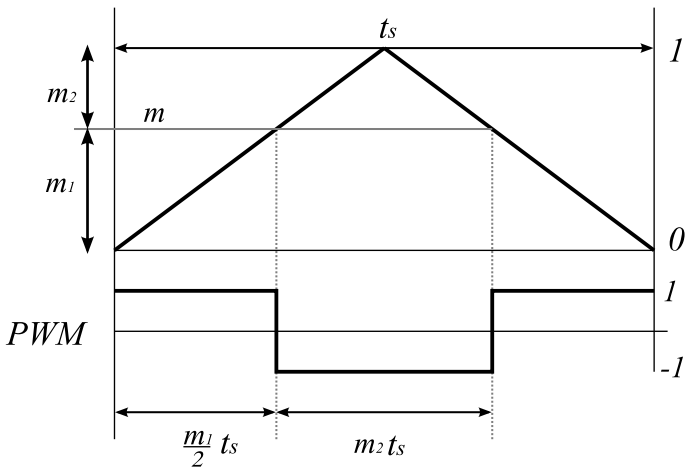

Figure 7: Timing of symmetrical PWM.

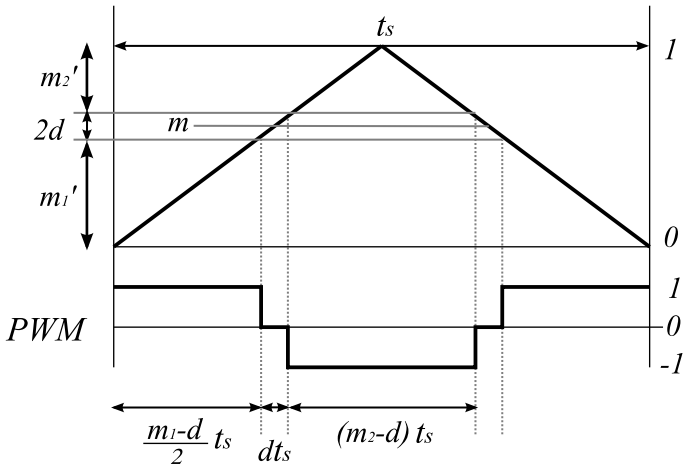

Figure 8: Timing of symmetrical PWM with blanking

Revision History:

Tutorials 1.0	First release

How to Contact Plexim:

☎	+41 44 533 51 00	Phone
	+41 44 533 51 01	Fax
✉	Plexim GmbH Technoparkstrasse 1 8005 Zurich Switzerland	Mail
@	`info@plexim.com`	Email
	`http://www.plexim.com`	Web

PLECS Tutorial