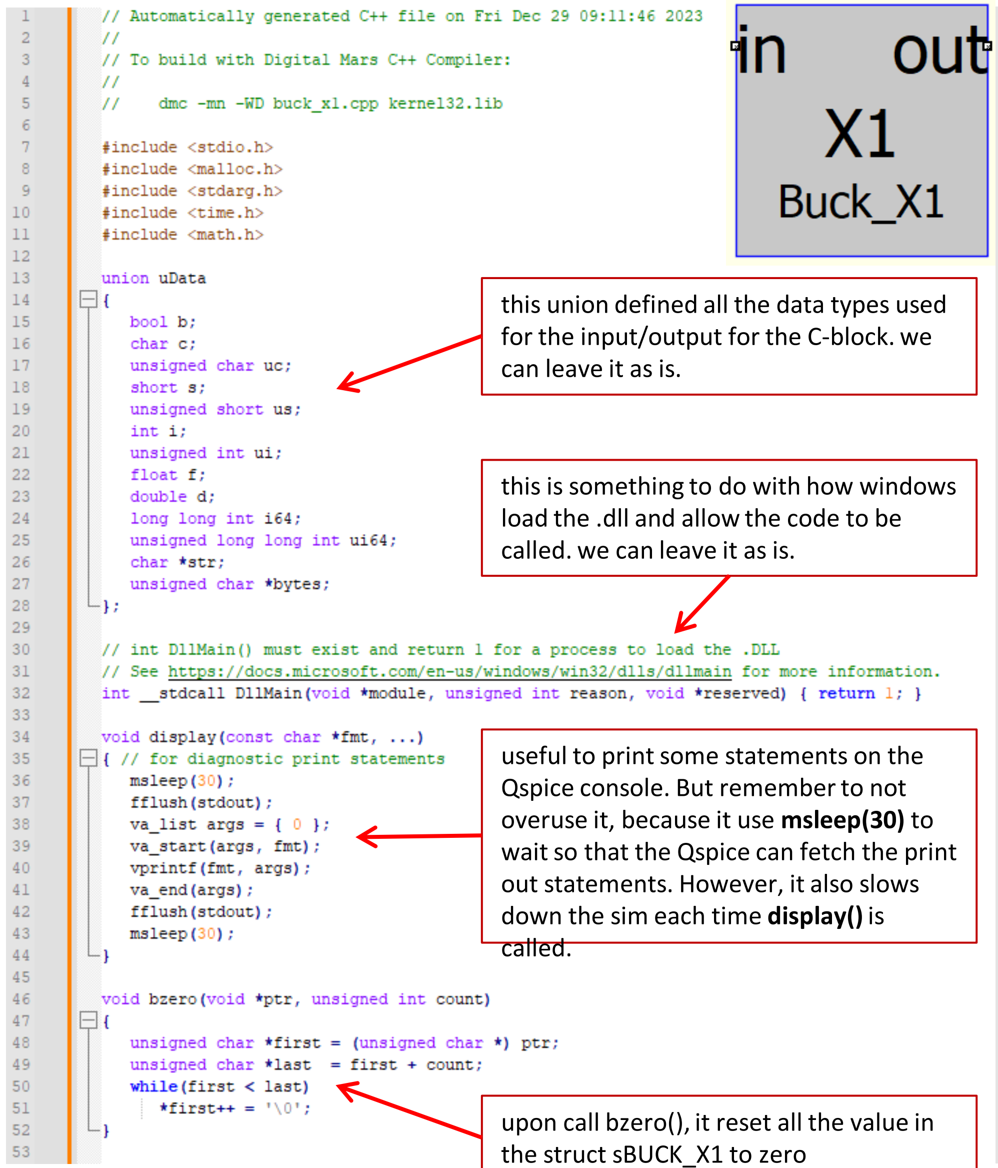
Three Complexity Levels of PWM Implementation for Digital Controller Simulation

By: Arief Noor Rahman

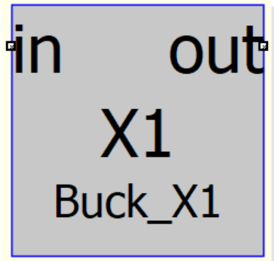
Background

- Qspice has great improvement over its predecessor (LTspice). My favorite feature is the added support for custom digital block (C and Verilog). Which allows more efficient and more direct implementation of various control, analysis, and signal processing algorithm (instead of using circuit/behavioral source or as post processing).
- This article provides a method on how to use C-block to achieve digital smps simulation as fast and as accurate as possible.
- Some background information about the C-block implementation as well as my own understanding of the interaction between main Spice solver with the C-block are provided.

Structure of C-block for Qspice



```
// Automatically generated C++ file on Fri Dec 29 09:11:46 2023
//
// To build with Digital Mars C++ Compiler:
//
//    dmc -mn -WD buck_x1.cpp kernel32.lib

#include <stdio.h>
#include <malloc.h>
#include <stdarg.h>
#include <time.h>
#include <math.h>

union uData
{
   bool b;
   char c;
   unsigned char uc;
   short s;
   unsigned short us;
   int i;
   unsigned int ui;
   float f;
   double d;
   long long int i64;
   unsigned long long int ui64;
   char *str;
   unsigned char *bytes;
};

// int DllMain() must exist and return 1 for a process to load the .DLL
// See https://docs.microsoft.com/en-us/windows/win32/dlls/dllmain for more information.
int __stdcall DllMain(void *module, unsigned int reason, void *reserved) { return 1; }

void display(const char *fmt, ...)
{ // for diagnostic print statements
   msleep(30);
   fflush(stdout);
   va_list args = { 0 };
   va_start(args, fmt);
   vprintf(fmt, args);
   va_end(args);
   fflush(stdout);
   msleep(30);
}

void bzero(void *ptr, unsigned int count)
{
   unsigned char *first = (unsigned char *) ptr;
   unsigned char *last  = first + count;
   while(first < last)
      *first++ = '\0';
}
```

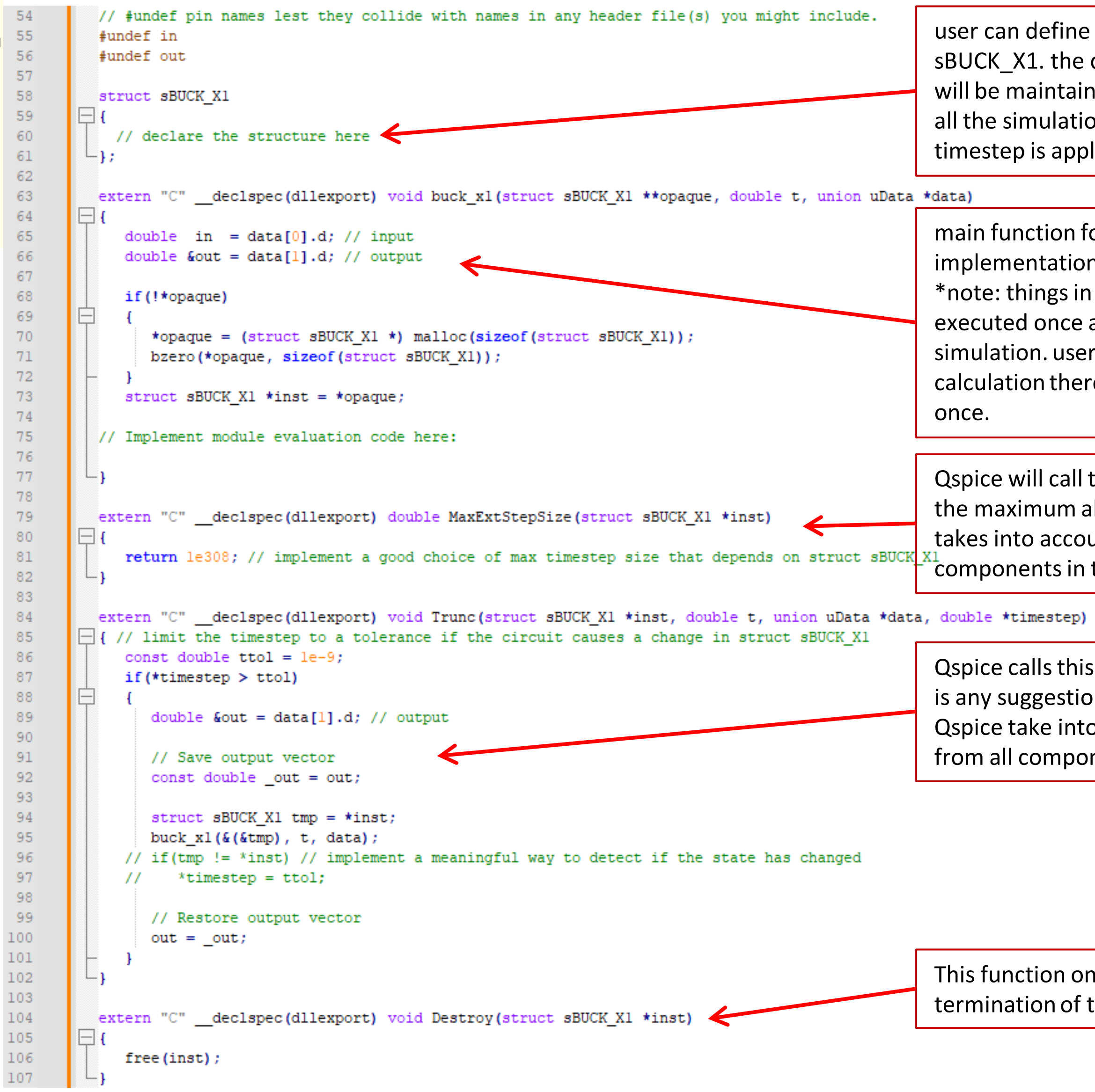


this union defined all the data types used for the input/output for the C-block. we can leave it as is.

this is something to do with how windows load the .dll and allow the code to be called. we can leave it as is.

useful to print some statements on the Qspice console. But remember to not overuse it, because it use **msleep(30)** to wait so that the Qspice can fetch the print out statements. However, it also slows down the sim each time **display()** is called.

upon call bzero(), it reset all the value in the struct sBUCK_X1 to zero



```
// #undef pin names lest they collide with names in any header file(s) you might include.
#undef in
#undef out

struct sBUCK_X1
{
  // declare the structure here
};

extern "C" __declspec(dllexport) void buck_x1(struct sBUCK_X1 **opaque, double t, union uData *data)
{
   double  in  = data[0].d; // input
   double &out = data[1].d; // output

   if(!*opaque)
   {
      *opaque = (struct sBUCK_X1 *) malloc(sizeof(struct sBUCK_X1));
      bzero(*opaque, sizeof(struct sBUCK_X1));
   }
   struct sBUCK_X1 *inst = *opaque;

// Implement module evaluation code here:

}

extern "C" __declspec(dllexport) double MaxExtStepSize(struct sBUCK_X1 *inst)
{
   return 1e308; // implement a good choice of max timestep size that depends on struct sBUCK_X1
}

extern "C" __declspec(dllexport) void Trunc(struct sBUCK_X1 *inst, double t, union uData *data, double *timestep)
{ // limit the timestep to a tolerance if the circuit causes a change in struct sBUCK_X1
   const double ttol = 1e-9;
   if(*timestep > ttol)
   {
      double &out = data[1].d; // output

      // Save output vector
      const double _out = out;

      struct sBUCK_X1 tmp = *inst;
      buck_x1(&(&tmp), t, data);
   // if(tmp != *inst) // implement a meaningful way to detect if the state has changed
   //    *timestep = ttol;

      // Restore output vector
      out = _out;
   }
}

extern "C" __declspec(dllexport) void Destroy(struct sBUCK_X1 *inst)
{
   free(inst);
}
```

user can define most of all data in struct sBUCK_X1. the data defined in sBUCK_X1 will be maintained by Qspice throughout all the simulation, even when negative timestep is applied.

main function for all the algorithm implementation.
*note: things in if(!opaque){} will be executed once at the beginning of the simulation. users can put some calculation there that just need to run once.

Qspice will call this function to determine the maximum allowable time step. Qspice takes into account all suggestion from all components in the simulation

Qspice calls this function to read if there is any suggestion for the next timestep. Qspice take into account all suggestion from all components in the simulation.

This function only called once upon the termination of the simulation.

Tricky Parts about C-block and Timing within Qspice

1. Qspice use variable timestep, furthermore upon nonlinearity event detection (i.e. transistor switching, or diode on DCM), Qspice can even apply negative timestep to ensure the exact timing of the non-linear event is properly captured.

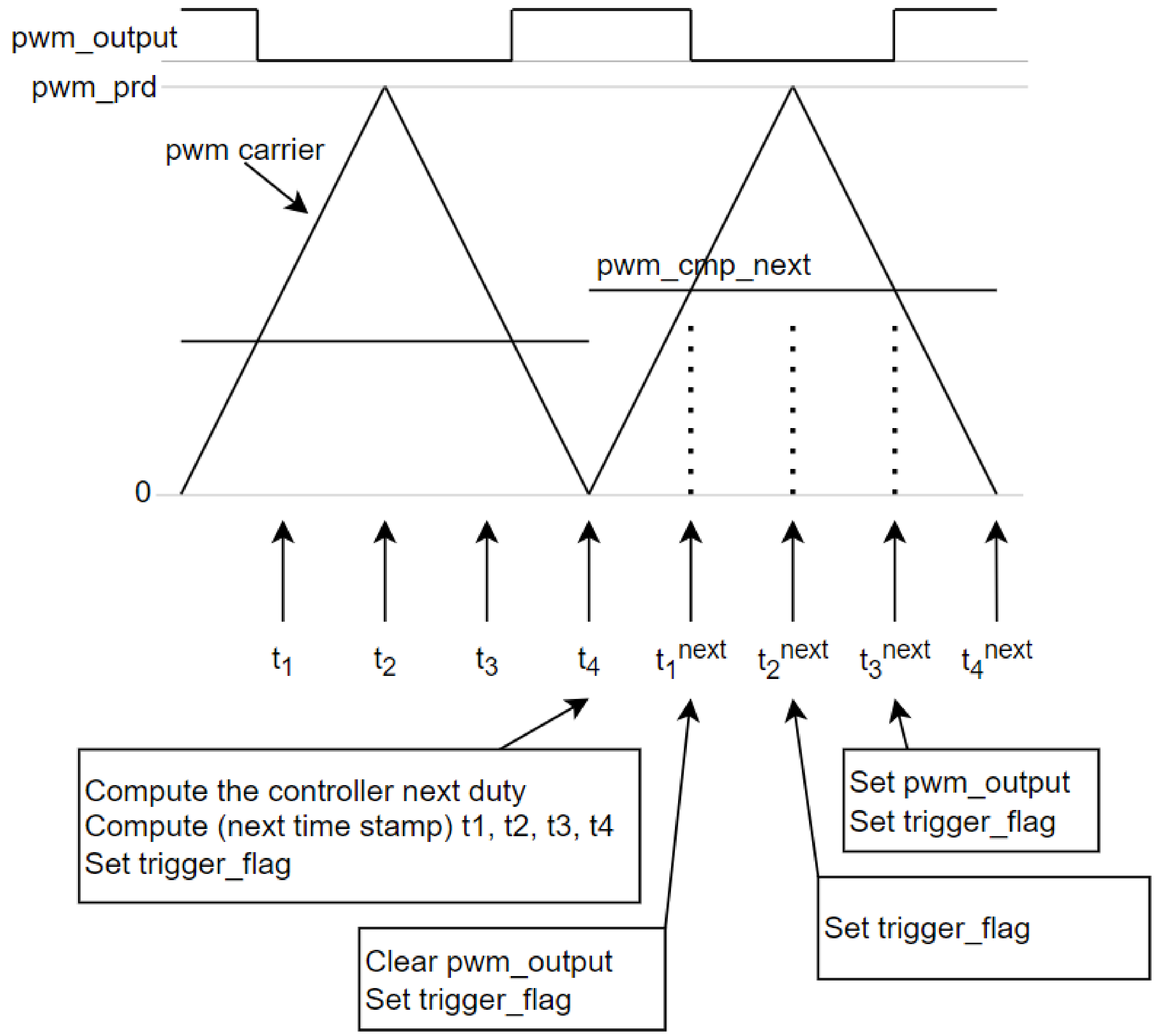
   This behavior can mess up with C-block as most control/signal processing algorithm are not designed with negative timestep. Where it can cause the user's algorithm to be called multitle time for the same simulation run time. To avoid this, ensure that the variables that is sensitive to simulation runtime must be defined within the struct sBuck_X1

2. Trunc() is used to provide suggestion of next timestep. The primary suggestion from the template is based on the working principle of Qspice own's variable timestep algorithm, where if the result of the next timestep induces a nonlinearity then revert back to the minimum timestep. After trying with minimum timestep for a few steps, then next timestep is double of the current timestep. The timestep will keep increasing until max timestep is reach or nonlinearity event is detected.

   To speed up the simulation, in case of digital control smps, we can exploit the characteristic of the digital pwm implementation where the duty cycle for the of the future switching period is defined by the last sampling period. Means we can set send the Qspice solver the next timestep suggestion to directly hit the next transistor switching time.

   *note: Qspice takes in the smallest maxtimestep and smallest timestep suggestion from all components in the simulation schematic

Timing Control Implementation Algorithm



Compute the controller next duty
Compute (next time stamp) t1, t2, t3, t4
Set trigger_flag

Clear pwm_output
Set trigger_flag

Set trigger_flag

Set pwm_output
Set trigger_flag

for output with deadtime, pwm_delay is generated by creating one additional time stamp when simulation time:
a. hit t1_next at t1_next + dtime
or
b. hit t2_next at t2_next + dtime

when simulation time hits t1_next + dtime or t2_next + dtime, the pwm_delay will copy the value of the pwm_output

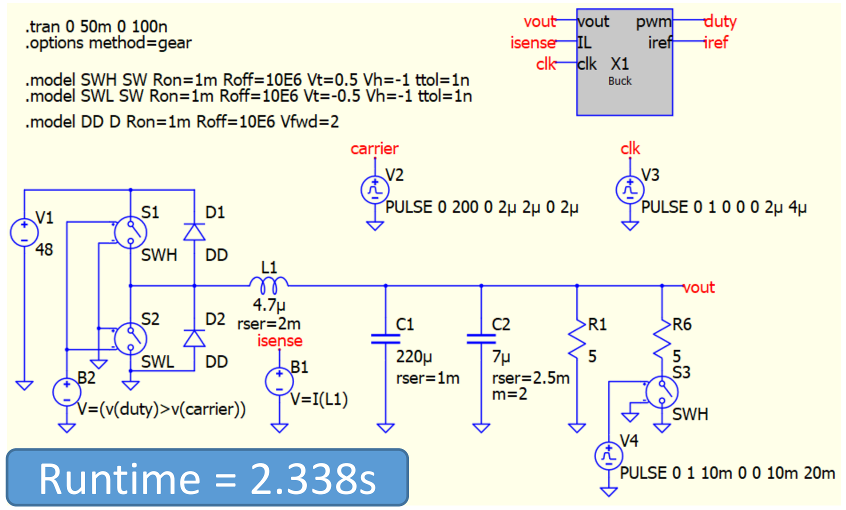
pwm_hi = pwm_output & pwm_delay
pwm_lo = !pwm_output & !pwm_delay

Trunc() will read t1, t2, t3, t4, and trigger_flag

It will set the next time_step suggestion for Qspice directly to the nearest future timing mark.
In case trigger_flag is set, if the nearest future time stamp is greater than timing tolerance, then next time_step will be set at timing tolerance.

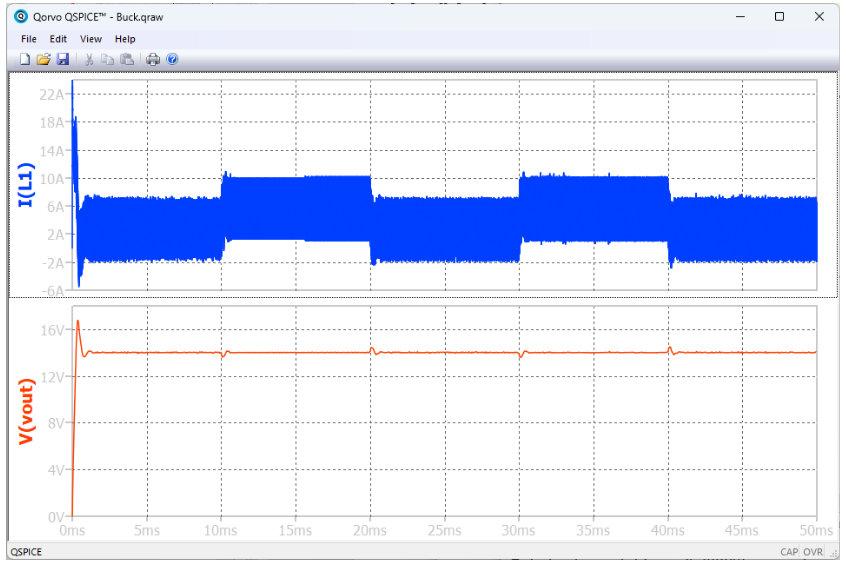
The three levels of complexity

Level 1 : C-block only for control algorithm



```
.tran 0 50m 0 100n
.options method=gear

.model SWH SW Ron=1m Roff=10E6 Vt=0.5 Vh=-1 ttol=1n
.model SWL SW Ron=1m Roff=10E6 Vt=-0.5 Vh=-1 ttol=1n
.model DD D Ron=1m Roff=10E6 Vfwd=2
```

Runtime = 2.338s



In this approach, C-block only performs the control algorithm that read analog feedback and send out duty cycle command. Sampling timing control for C-block is provided by the **clk (V3)** and the pwm pulse output is provided by **carrier (V2)** and **comparator (B2)**.

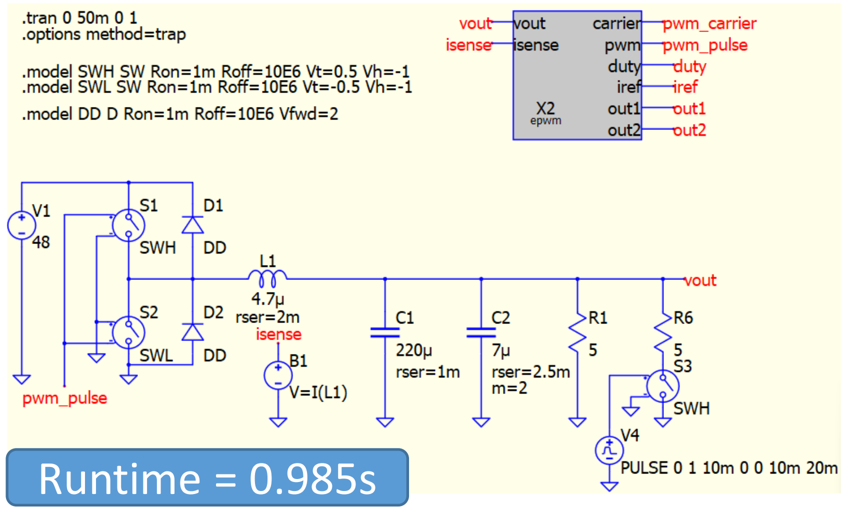
Note 1: Model definition for SWH and SWL have parameter **ttol=1n** to force Qspice to iterate around the exact switching event with the accuracy down to 1ns. Additionally, Qspice also conduct some iteration around the peak and valley of the carrier triangle waveform and at the rise/fall time of clk source.

Note 2: Simulation exhibit some strange unrealistic oscillation, thus the solver changed to gear which have oscillation damping behavior. The oscillation still occurs in the simulation.

Note 3: This method is the simplest, however the multiple timestep iteration around the discontinuity event can increase the total simulation time, especially for complex simulation with higher number of switches.

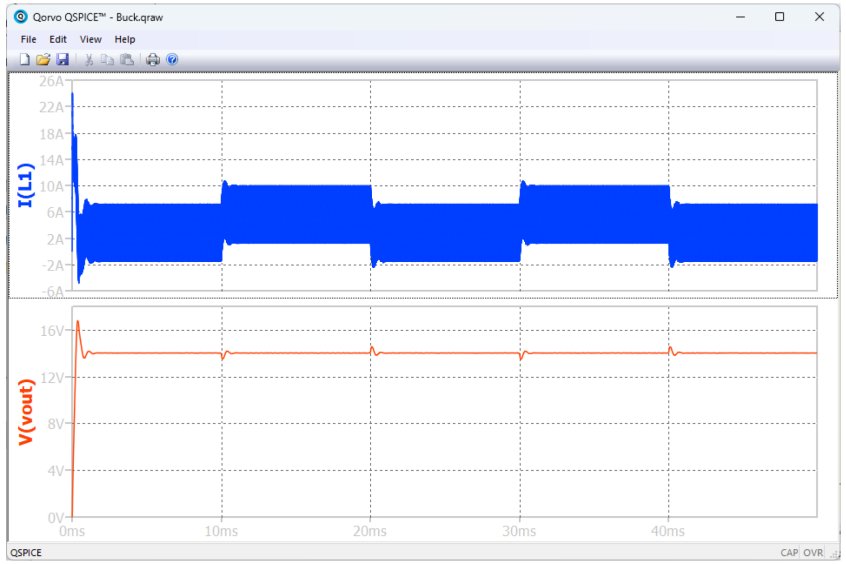
The three levels of complexity

Level 2 : C-block for control and basic pwm generation



```
.tran 0 50m 0 1
.options method=trap

.model SWH SW Ron=1m Roff=10E6 Vt=0.5 Vh=-1
.model SWL SW Ron=1m Roff=10E6 Vt=-0.5 Vh=-1

.model DD D Ron=1m Roff=10E6 Vfwd=2
```

Runtime = 0.985s



*In this design, PWM generation method follow the method used by TI C2000 MCU family.

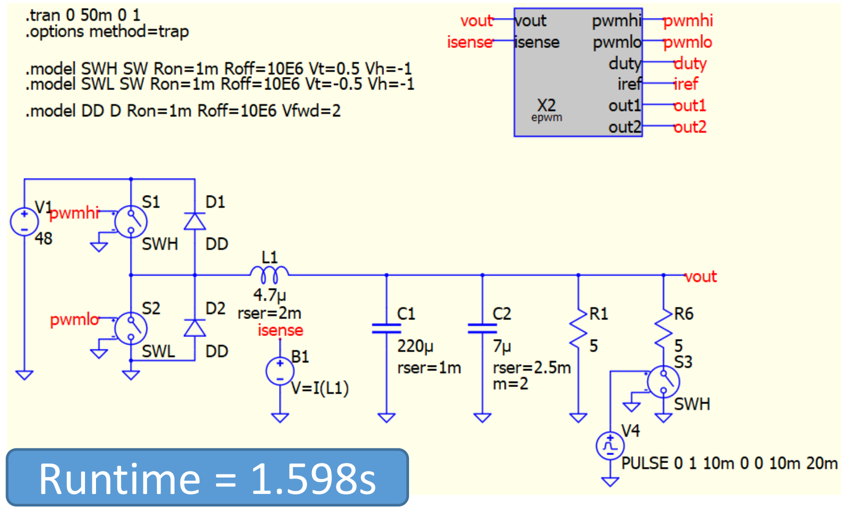
For this implementation, the PWM generation, sampling timing, and control algorithm are all implemented in C-block. While for a more basic implementation the PWM generation and sampling timing can be designed using an actual triangle wave within C-block, in this implementation we skipped the triangle waveform and directly compute the future discontinuity event.

Benefit of this approach is reduced number of timestep iteration especially around the discontinuity.

Disadvantage is a small added complexity of computing the and implementing the future discontinuity event algorithm.

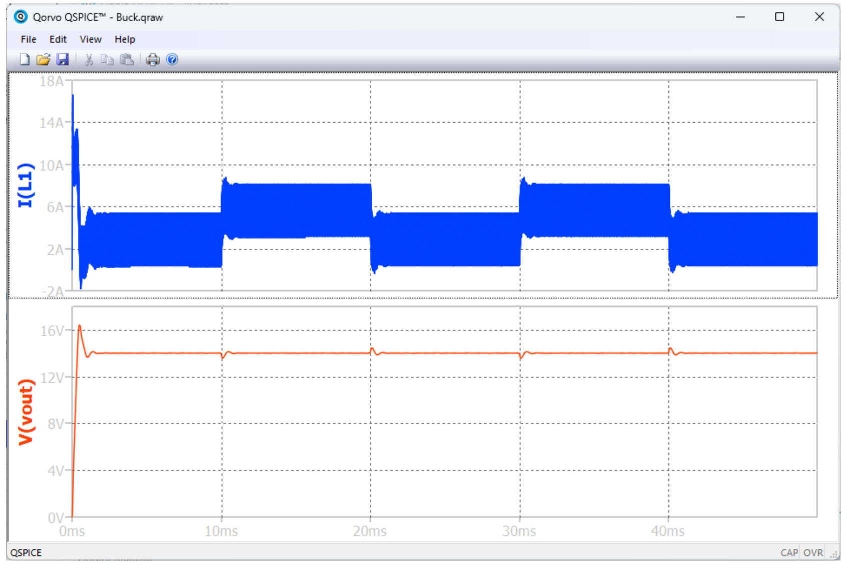
The three levels of complexity

Level 3 : C-block for control and PWM with deadtime



```
.tran 0 50m 0 1
.options method=trap

.model SWH SW Ron=1m Roff=10E6 Vt=0.5 Vh=-1
.model SWL SW Ron=1m Roff=10E6 Vt=-0.5 Vh=-1

.model DD D Ron=1m Roff=10E6 Vfwd=2
```

Runtime = 1.598s

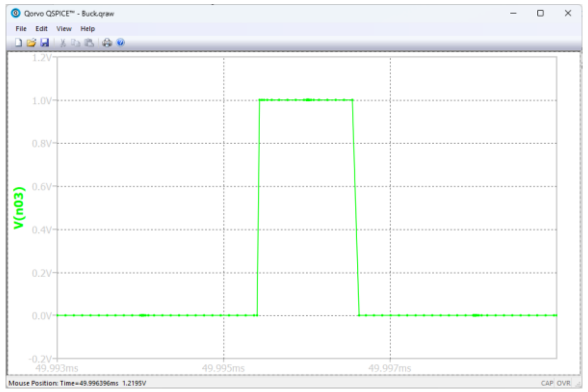


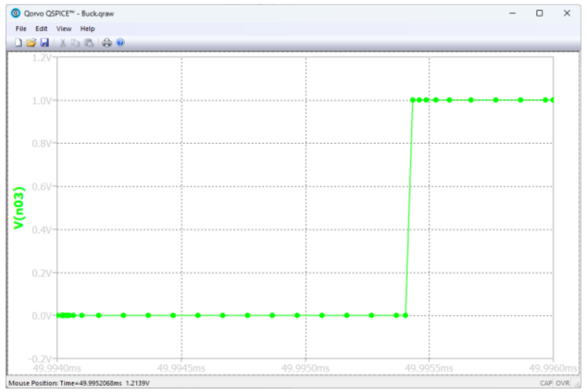
The level 3 implementation extends the details of the implementation on level 2 further by implementing dead-time algorithm.

*In this design, PWM generation method follow the method used by TI C2000 MCU family.

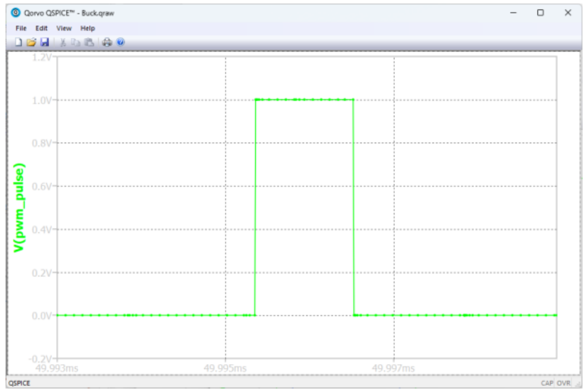
PWM signal comparison (discontinuity event capture)

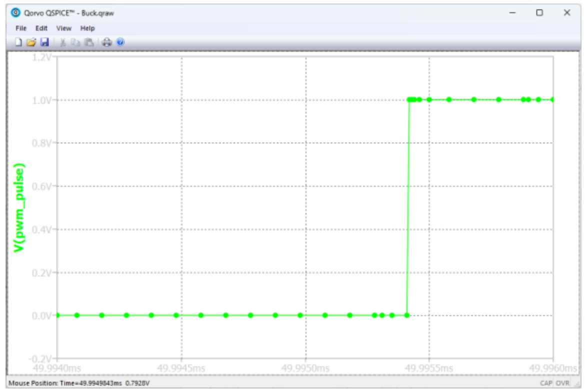
Level 1 PWM



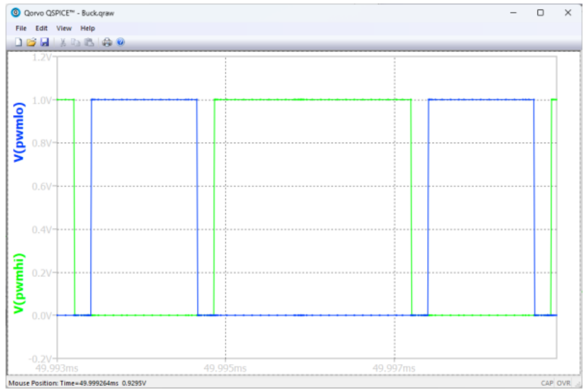


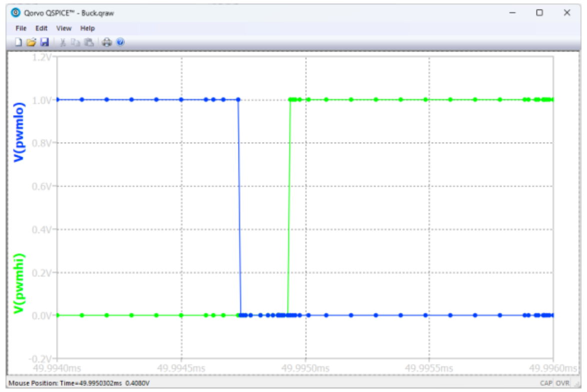
Level 2 PWM





Level 3 PWM





The key for simulation speed improvement is by simply reducing the effort for Qspice timing solver to accurately capture the exact discontinuity event timestamp.

Level 2 PWM clearly have significantly less simulation points than Level 1 PWM.

While the Level 3 clearly has more time stamp than Level 2 due to the deadtime generation, however the generated timestamp is still less than Level 1 PWM. Thus the shorter time to complete the simulation.

Summary

1. The high simulation accuracy and timing accuracy can be achieved by using the Trunc() feature.
2. For the same simulation condition (without deadtime), the timing algorithm can reduce the simulation time down by 58%. Even with deadtime (highest accuracy) the simulation time is 32% faster than using the Qspice discontinuity detection.
3. Another benefit of the timing control approach is more accurate overall simulation as indicated by no simulation artifact to be visible on the output voltage and inductor current waveforms.