Application Report

Intelligent Stepper Motor Driver Control

Jose Quiñones

ABSTRACT

This document is provided as a supplement to the data sheets of the devices in Table 1-1. The table details a technique to improve real-time control of the integrated indexer of these bipolar stepper motor drivers, while obtaining programmable acceleration and deceleration profiles, speed control and position control with a MSP430 microcontroller.

Table 1-1. Integrated Stepper Motor Drivers

Device	Recommended Max. Voltage (V)	Full Scale Current (A)	Rdson (mohm) LS+HS FET typical at 25°C	Control	Stall Detect	Smart Tune	Classification
DRV8434S	48	2.5	330	SPI	yes	Programmable	Industrial
DRV8434A	48	2.5	330	GPIO	yes	Fixed	Industrial
DRV8434	48	2.5	330	GPIO	no	Fixed	Industrial
DRV8428	33	1	1500	GPIO	no	Programmable	Industrial
DRV8426	33	1.5	900	GPIO	no	Programmable	Industrial
DRV8425	33	2	550	GPIO	no	Programmable	Industrial
DRV8424	33	2.5	330	GPIO	no	Programmable	Industrial
DRV8436	48	1.5	900	GPIO	no	Programmable	Industrial
DRV8889-Q1	45	1.5	900	SPI	yes	Programmable	Automotive
DRV8899-Q1	45	1	1200	SPI	no	Programmable	Automotive

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction and Problem Statement

Driving a stepper motor can be a daunting task. Whereas, providing a voltage at the terminals of a DC motor causes immediate rotation, on a stepper motor, careful magnetic field commutation must be applied in order to obtain the same behavior. In the not so distant past, said electromagnetic commutation was achieved by coding powerful microprocessors to coordinate the phase and current information administered into the power stage.

With the advent of high integration on monolithic integrated circuits, it became simpler to take into hardware all the blocks once generated through code. A stand alone IC could now control even the most intricate subjects such as phase commutation and microstepping without the need of precious microcontroller resources.

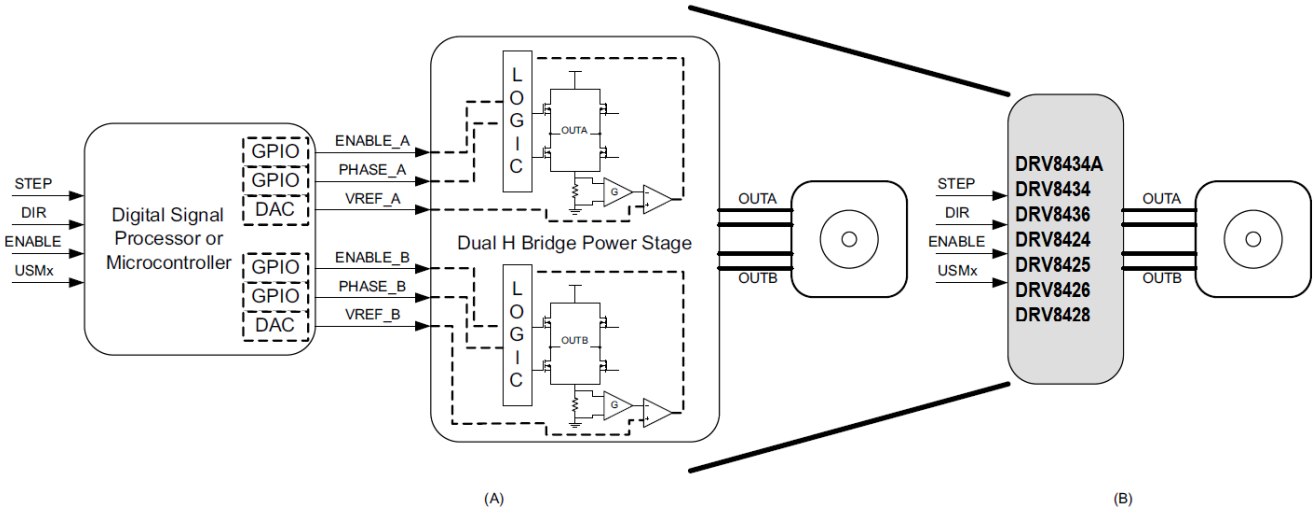


Figure 1-1. Stepper Control Logic and Power Stage

Figure 1-1 shows the level of integration which can be obtained when the code inside of a microcontroller, and in charge of causing stepper commutation, is concatenated along with the power stage into a single chip solution. Notice that in both scenarios a series of simple control signals exist. A STEP pulse is used to generate steps or microsteps; a DIR signal defines the direction of rotation; the ENABLE line determines whether the power stage is enabled or not; and the User Mode bits are used to select a degree of microstepping.

Controlling a stepper, however, can still benefit from the usage of a microcontroller. Tasks such as speed and position control, acceleration and deceleration, homing, etc. still require accuracy and precision which a microcontroller can easily supply. The question we must ask is: Will the application processing unit be asked to compute all the parameters related to stepper motion, or will a smaller and more cost economical microcontroller be used to tackle the tasks at hand?

Using a smaller microcontroller with each driver to perform the aforementioned tasks is advantageous if numerous steppers are to be controlled. In this fashion, the application processor can utilize its real time resources to properly coordinate application intensive aspects, while the small microcontrollers deal with the intricacies of controlling the stepper motors.

This application note details an implementation using an MSP430F2132 microcontroller and a DRV8434/24 device which has an internal indexer bipolar stepper microstepping power stage. Combined, they form a module capable of receiving commands from a controller through an I^2C bus, and which will then undertake all the actions to control the stepper motor both in speed and position. In order to best utilize the available resources, a series of GPIO terminals were added, which will provide extra functionality to the main processor. Figure 1-2 shows a block diagram of the proposed implementation.

While the implementation considers GPIO controlled stepper drivers, it can be adapted to SPI controlled drivers. Some of the newer TI stepper drivers have integrated stall detection which can eliminate the need for external homing and end of travel sensors as well as safeguard the electrical and mechanical systems from wear and tear. Stepper drivers such as the DRV8434S offer stall detection using SPI control and digital implementation whereas the DRV8434A offer stall detection using GPIO control and analog implementation. In addition newer TI

stepper drivers offer integrated current sensing eliminating the need for external current sense resistors, not only reducing cost and PCB space but also cutting power loss and heat. Furthermore, integrated smart tune technology in the newer devices increases efficiency and minimize audible noise from the stepper motor.

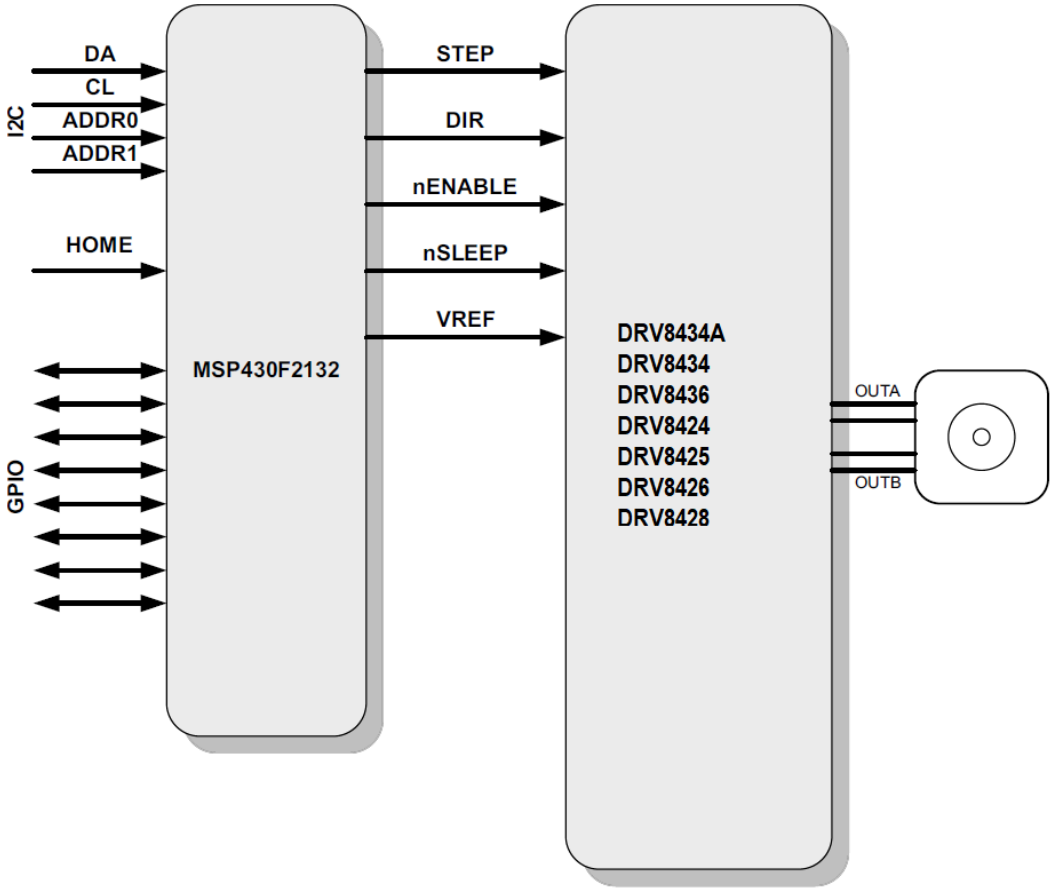


Figure 1-2. Intelligent Stepper Controller Block Diagram

2 Stepper Motor Control High Level Functions

2.1 STEP Actuation: Acceleration, Speed Control and Deceleration Profiles

Stepper motors offer a means to achieve speed control without the usage of closed loop mechanisms such as shaft encoders or resolvers. On a microstepping internal indexer driver, this open loop control is obtained by modulating the frequency at the STEP input. Each pulse at the STEP input, becomes a mechanical step motion at the stepper motor. Hence, it is safe to say that since we know what frequency we are applying at the STEP input, we then know the actual step rate the stepper motor is moving at. This will hold true as long as the right parametric values, such as current, voltage and torque, are maintained within reasonable levels throughout the application's operation.

Unfortunately, we cannot just apply any frequency or step rate to any given stepper motor. Due to the mechanisms behind the revolving magnetic field at the stator and the permanent magnet at the rotor, a stepper motor can only start moving if the requested speed is smaller than a parameter given by the motor's manufacturer and referred to as the starting frequency (denominated FS). For example, if the FS for a particular stepper motor is 300 steps per second (SPS), it will most likely not be possible to start the motor at a frequency of 400 SPS.

Since the application may require speed rates larger than the FS, it is then very important to subject the motor commutation through an acceleration profile which starts at a speed rate lower than its maximum FS and increases speed accordingly until reaching the desired speed.

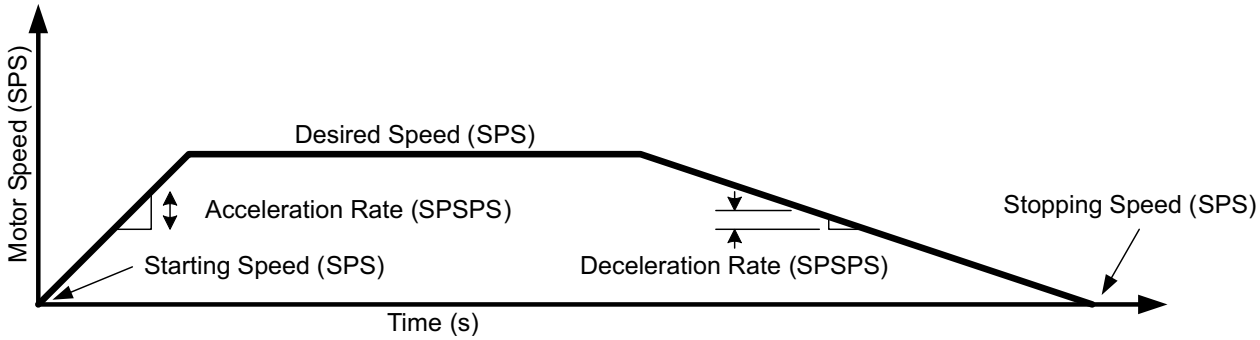


Figure 2-1. Typical Stepper Acceleration and Deceleration Profile

Figure 2-1 shows a typical acceleration and deceleration profile where:

Starting Speed is a STEP frequency lower than the motor's rated FS at which the motor will start moving. Measured in steps per second (SPS), where STEPS refers to full steps.

Acceleration Rate is a factor of how much the STEP frequency will be increased on a per second basis. Measured in steps per second per second (SPSPS).

Desired Speed is the STEP frequency the application requires the motor to move at. It marks the STEP frequency at which the acceleration profile concludes. Measured in steps per second.

Deceleration Speed is a factor of how much the STEP frequency will be decreased on a per second basis. Measured in steps per second per second (SPSPS).

Stopping Speed is the STEP frequency at which the deceleration profile and the motor will be stopped. In this application note, stopping speed is taken to be the same as the starting speed. Measured in steps per second.

2.2 Accelerating the Motor

The start stepper command starts by issuing steps at a starting frequency denoted by the StartingSpeed variable. A timer must be used to generate STEP pulses at such frequency. On this application note, Timer A0.2 was used to set the STEP signal and Timer A0.0 was used to clear the same signal. The pulse width is 32 clock pulses wide which translates to 2 µs. Since the DRV8825 requires STEP pulses at least 1-µs wide, this implementation results in a legal pulse generation.

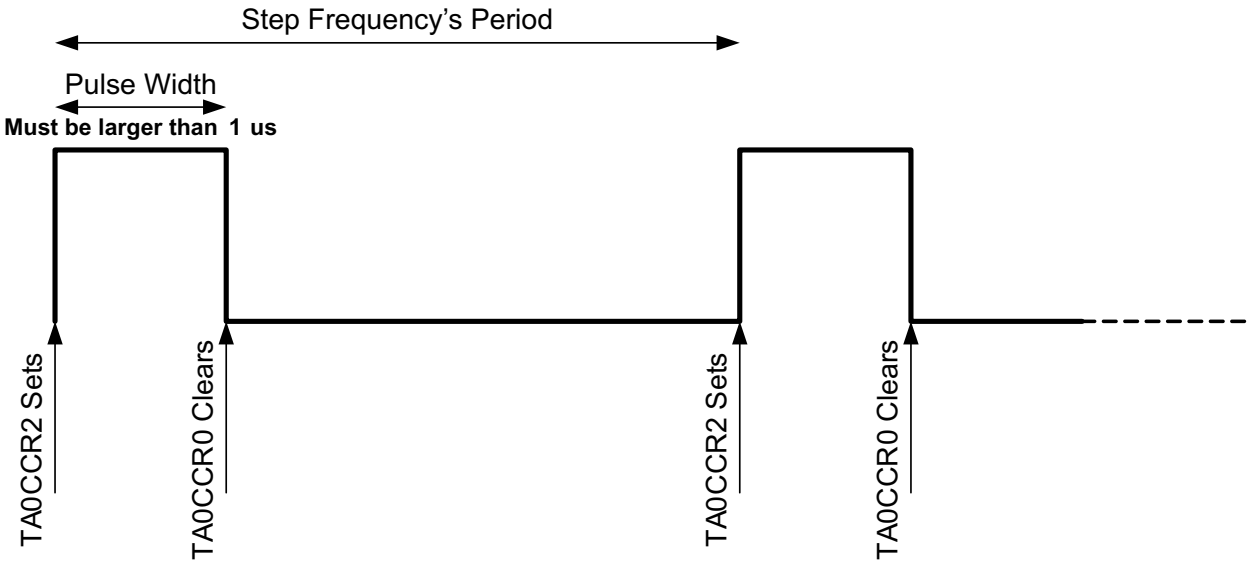


Figure 2-2. Timer A0 Used to Generate STEP Pulses

The interrupt subroutine for Timer A0.2 is in charge of programming the Timer A0.0 hardware reset, as well as programming itself in order to schedule the next step generation.

A second timer function is needed to generate the acceleration portion of the speed control profile. The microcontroller selected for this application only had two available timers so as will be explained shortly, a trade off had to be made.

Timer A1.1 was used along with timer A1.0 to generate the PWM output which would be used to generate an analog to digital converter, by adding an RC filter, to drive the DRV8825's VREF analog input pin. With this technique, it is then possible to control stepper current in real time.

Although all timer resources seem to be depleted, there is still an aspect of timer A1 we can utilize. If we code the PWM generator to count from decimal 0 to decimal 249, we will have 250 time units. At 16 MHz, each timer unit is equal to 62.5 ns with 250 of them accounting for 15.625 µs. By further dividing this result by 8, we can get a fairly useful time base of 125 µs, or 4000 timer ticks per second.

What this means is we can increase the stepper speed rate up to 4000 times per second. A simple mathematical equation can then be used to compute both the acceleration time interval and the acceleration increment parameter. The code snippet below shows the equations used to compute both aspects of the acceleration parameter:

```
void AccelTimeCompute(unsigned int AccelDecelRate)
{
  if (AccelDecelRate <= 4000)
    {
    AccelerationTime = 4000/AccelDecelRate;
    AccelerationIncrease = 1;
    }
  else
    {
    AccelerationTime = 1;
    AccelerationIncrease = AccelDecelRate/4000;
    }
  tmpAccelerationTime = AccelerationTime;
}
```

Figure 2-3. Function Computing Time Interval and Acceleration Increase Parameters for Acceleration or Deceleration Rate

Figure 2-4 shows the flowchart of the state machine in charge of coordinating the acceleration. As can be seen, the command to start the motor will configure all the parameters required for motion, including the call to the AccelTimeCompute function.

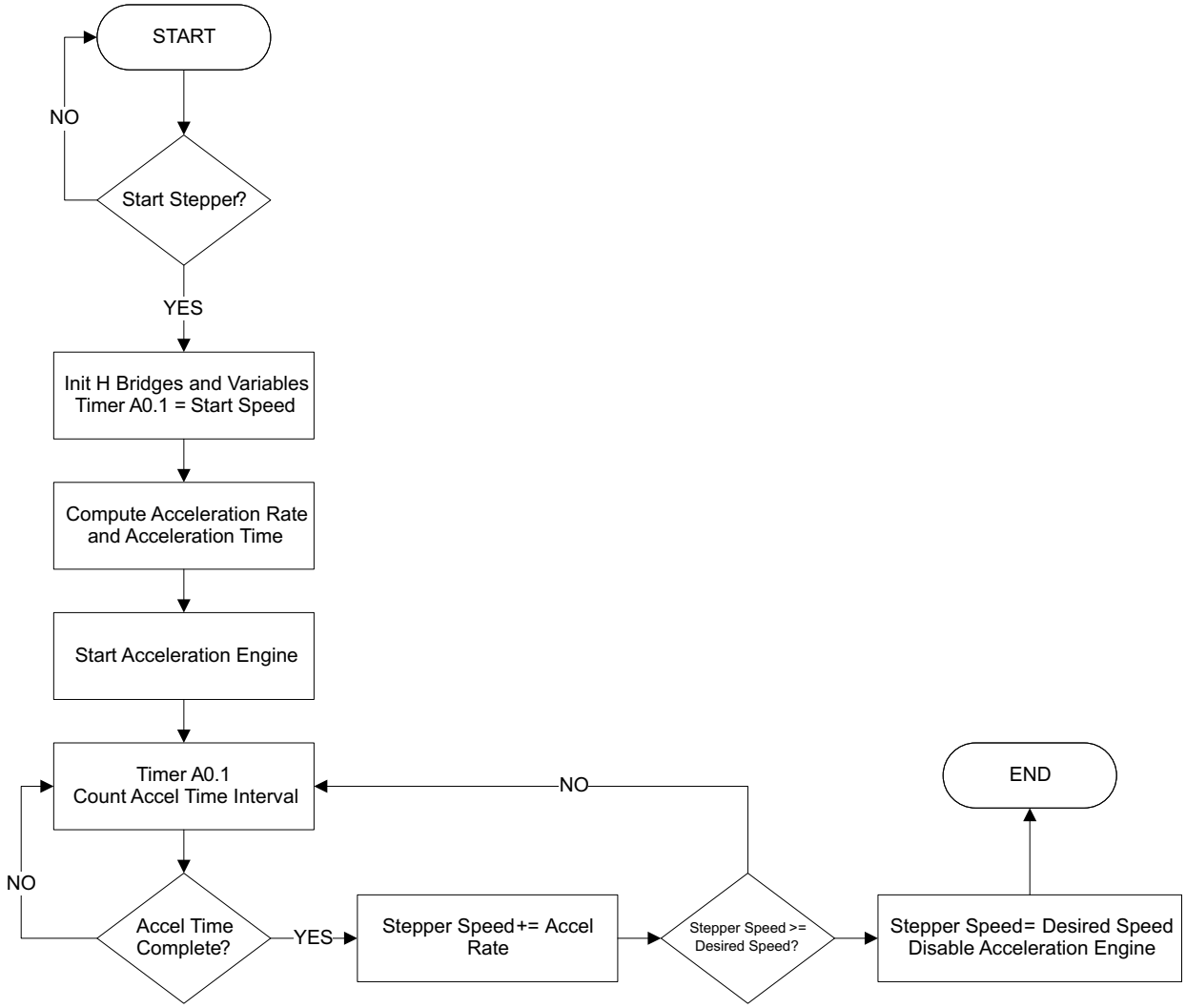


Figure 2-4. Stepper Speed Acceleration Flowchart

The actual speed modification takes place inside the AccelDecel function which gets called from within the main execution which is in turn enabled within the timer A1.0 interrupt service routine (ISR). It is important for the AccelDecel function not to be called from within inside the ISR as this impacts real time operation. Instead, inside the ISR the microcontroller sleep mode is disabled, which allows execution to resume from the main loop.

```
int main()
{

//ALL INITIALIZATION OCCURS HERE

__bis_SR_register(GIE);                  // Enable all Interrupts
while (1)
 {
 _BIS_SR(CPUOFF);                  // Enter LPM0
 AccelDecel();
 }
}
```

Figure 2-5. Main Function Calls the AccelDecel Code Whenever Code Inside of an ISR Removes the Micro from Sleep Mode

```
#pragma vector=TIMER1_A0_VECTOR
__interrupt void Timer1_A0(void)
{
  TimerA0Count += 1;
  if (TimerA0Count == ATBCount)
    {
    TimerA0Count = 0;
    if (tmpAccelerationTime == 0)
      {
      tmpAccelerationTime = AccelerationTime;
       _BIC_SR_IRQ(CPUOFF);              // Clear LPM0 - jumps to AccelDecel();
      }
    tmpAccelerationTime -= 1;
    }
}
```

Figure 2-6. Timer A1.0 ISR Disables Sleep Mode Once an Acceleration Tick is to be Executed

```
void AccelDecel(void)
{
  switch (AccelerationState)
 {
 case (NOACC):
  break;
 case (ACCEL):
  ActualStepperSpeed += AccelerationIncrease;
  if (ActualStepperSpeed >= DesiredStepperSpeed)
   {
    ActualStepperSpeed = DesiredStepperSpeed;
    AccelerationState = NOACC;
    TA1CCTL0 &= ~CCIE;       //DISABLE 250 us coordinator interrupt on TA1.0
   }
  SpeedCompute(ActualStepperSpeed);

  break;
 case (DECEL):
  ActualStepperSpeed -= AccelerationIncrease;
  if (ActualStepperSpeed <= DesiredStepperSpeed)
   {
    ActualStepperSpeed = DesiredStepperSpeed;
    AccelerationState = NOACC;
    TA1CCTL0 &= ~CCIE;       //DISABLE 250 us coordinator interrupt on TA1.0
   }
  SpeedCompute(ActualStepperSpeed);

  break;
 case (STOP):
  ActualStepperSpeed -= AccelerationIncrease;
  if (ActualStepperSpeed <= StartStepperSpeed)
   {
    ActualStepperSpeed = StartStepperSpeed;
    AccelerationState = NOACC;
    TA1CCTL0 &= ~CCIE;       //DISABLE 250 us coordinator interrupt on TA1.0
    ExecutedSteps = NumberOfSteps - 1;
   }
  SpeedCompute(ActualStepperSpeed);
  break;
 }
}
```

Figure 2-7. AccelDecel Function is State Machine Code in Charge of Modifying Stepper Speed According to Programmed Acceleration or Deceleration Profile

2.3 Stepper Speed

Stepper speed is carefully controlled by using Timer A1.0 and A1.2 as detailed previously. How the timer is being clocked is important to understand, as the 16- bit timer by itself is not capable of reproducing all the possible speeds applications around this topology may try to run at. For example, while the Timer A1 is being clocked at 16 MHz, fast stepping rates are easily obtained, while slow speeds cannot be generated.

As a result, it is imperative to modulate the clock speed by subdividing it. Luckily Timer A has a series of dividing factors which allow the timer clocking speed to be subdivided from 16 MHz down to as slow as 2 MHz.

The equation to compute how many clock cycles are needed to reproduce a particular stepping rate is:

$$\#\ \text{Clock Cycles} = \text{TIMER CLOCK FREQUENCY} / \text{STEP\_RATE in Hz} \quad (1)$$

A quick analysis on the results provided by the previous equation showed the stepping rate segregations on a per clock cycle count basis. If for a given timer clock frequency the resulting number of clock cycles was larger than 65535 then a further subdivision factor would be needed to reduce the clock count from saturating the timer register.

The function SpeedCompute is in charge of transforming a stepping rate in Hz into clock count. Depending on the speed, the timing source for Timer A1 is sub divided to slow down its incremental rate. If the requested stepping rate is less than 31, the function returns false in which case no rate is generated.

Although it may be desirable to obtain stepping rates lower than 31 full steps, this can still be achieved by utilizing a faster speed and configuring the DRV8824/25 device to operate in 32 degrees of microstepping. For example, if the programmed DesiredSpeed is 32 SPS, and the DRV8824/25 is configured to operate in 32 degrees of microstepping, the stepper will move 1 full step per second.

```
bool SpeedCompute(unsigned int MotorSpeedInHz)
{
   if (MotorSpeedInHz < 31 )
   {
     return false;
   }
   else if (MotorSpeedInHz < 61 )
   {
     ClockConfigure(8);
     StepperSpeedTMR = 2000000 / MotorSpeedInHz;
     return true;
   }
   else if (MotorSpeedInHz < 123)
   {
     ClockConfigure(4);
     StepperSpeedTMR = 4000000 / MotorSpeedInHz;
     return true;
   }
   else if (MotorSpeedInHz < 245)
   {
     ClockConfigure(2);
     StepperSpeedTMR = 8000000 / MotorSpeedInHz;
     return true;
   }
   else
   {
     ClockConfigure(1);
     StepperSpeedTMR = 16000000 / MotorSpeedInHz;
     return true;
   }
}
```

Figure 2-8. SpeedCompute Transform the Stepping Rate in Hz into a Clock Count Timer A1.1 - Can Use to Generate Accurate Timing Information

```
void ClockConfigure(char Divider)
{
  int tempTA0CTL;
  tempTA0CTL = TA0CTL;
  switch (Divider)
  {
  case 1:
  tempTA0CTL &= ~( BIT7 + BIT6);
  break;

  case 2:
  tempTA0CTL &= ~( BIT7 + BIT6);
  tempTA0CTL |= TA0_ID_DIV2;
  break;

  case 4:
  tempTA0CTL &= ~( BIT7 + BIT6);
  tempTA0CTL |= TA0_ID_DIV4;
  break;

  case 8:
  tempTA0CTL &= ~( BIT7 + BIT6);
  tempTA0CTL |= TA0_ID_DIV8;
  break;
  }
  TA0CTL = tempTA0CTL;
}
```

Figure 2-9. Function ClockConfigure is Called Within the SpeedCompute Function to Modify the Divider Affecting Timer A1 Clock Speed

The functions SpeedCompute and ClockConfigure are called to set the StartSpeed, to set the new stepper speed on a per acceleration click occurrence and then when the DesiredSpeed is met. Once the programmed DesiredSpeed target is reached the stepper motor maintains said speed until a deceleration profile is commanded.

2.4 Decelerating the Motor

The concept of a deceleration profile the motor is virtually identical to the acceleration profile implementation except that instead of increasing stepper speed, speed is decreased.

There are two instances in which a deceleration profile may be employed: during SpeedUpdate or during StepperStop commands. In this implementation, the stepper deceleration profile is finished either when the start speed is reached or when the programmed number of steps is executed.

Also, it was chosen for the deceleration rate to hold a different memory location such that a non symmetric acceleration/deceleration profile, such as the one portrayed in Figure 2-1, could be employed.

The SpeedCompute and ClockConfigure are used in the same fashion as they were used during the acceleration portion of the motion control profile. However, the state machine, or the AccelDecel function, is configured to execute the DECEL portion of code.

2.5 Speed Change

During run time, it is possible to change the speed and incur in further acceleration or deceleration as deemed by the application. When the command SpeedUpdate is issued, the firmware checks to see whether the new DesiredSpeed is larger or smaller than the ActualStepperSpeed. The result of this operation configures the Acceleration Deceleration engine accordingly.

2.6 Position Control: Number Of Steps

The closed loop capabilities of a stepper are not limited to accurately controlling the speed. Since in essence the controller is counting the very same steps which are being generated, knowledge of step position is continuously being updated. Every time a step is executed, the variable StepPosition is updated according to step rotation direction. For example, if the motor is rotating clockwise (DIR = HI), the StepPosition variable is incremented, whereas if the motor is rotating counterclockwise (DIR = LO), then the StepPosition variable is decremented.

Note

The notions of motor rotation as clockwise or counterclockwise are directly dependent to how the motor was wired and the motor itself.

The variable StepPosition can be read in real time to obtain information as to where the motor is at any given point in time.

On this application note, however, the controller is always operating in Position Control mode. The variable NumberOfSteps is programmed to hold the entire number of steps which will be executed. Since this is a 32-bit variable, the total number of steps could be a significantly large number, in which case the motor would in essence be operating in free running mode. However, if the NumberOfSteps count is small, then the motor will stop once the total NumbeOfSteps have been executed.

The previous mechanism is achieved by counting how many steps have been executed and comparing this number to the NumberOfSteps command. This is taken care of on the Timer A0.2 ISR as shown in Figure 2-10.

Notice this ISR is in charge of a few tasks we have described earlier and which are of crucial importance to generating accurate stepping information. These tasks are:

1. Generation of the STEP pulse by configuring when the STEP clear takes place. Timer A0.0 is in charge of returning the STEP signal to LO.
2. Configuring the TIMERA0.2 to generate the next step according to the current step rate, or StepperSpeedTMR, which is the timer equivalent of the current rate in Hz.
3. Determining whether the current generated step is the last step to execute. This happens when ExecutedSteps is equal to NumberOfSteps.
4. If the current generated step is equal to StepsToStop value, then the engine is commanded to start a deceleration profile.
5. Update StepPosition according to motor direction.

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer0_A1(void)
{

  switch (TA0IV)
    {
    case TA0CCR1_CCIFG_SET:
    break;
    case TA0CCR2_CCIFG_SET:

    TA0CCR0 = TA0CCR2 + StepPulseWidth;              //2 us at 16 MHz
    TA0CCR2 += StepperSpeedTMR;

    ExecutedSteps += 1;

    if (ExecutedSteps == NumberOfSteps)
      {
      TA0CCTL2 &= ~(CCIE + BIT5 + BIT6 + BIT7);      //Disable Pulse Generation
      }
    else if (ExecutedSteps == StepsToStop)
      {
      AccelTimeCompute(DecelerationRate);
      AccelerationState = STOP;
      TA1CCTL0 |= CCIE;                        //ENABLE 250 us coordinator interrupt on TA1.0
      }

    if (P3IN && DIR)
      {
      StepPosition += 1;
      }
    else
      {
      StepPosition -=1;
      }
    ReadTable[0] = (StepPosition & 0xFF00) >> 8;
    ReadTable[1] = StepPosition & 0xFF;
    break;
  case TA0IFG_SET:
    break;
    }
}
```

Figure 2-10. Timer A0.2 ISR

2.7 Homing the Stepper

The concept of preserving stepper position is in essence flawed if we do not know the system's start position. This is true for any motor system in which the closed loop feedback is relative, versus absolute. As a result, it is important we start counting our steps from a known position. Said position is often referred to as HOME.

The application note incorporates a HOME sensor input which is flexible enough to accommodate both sensor polarities (e.g. asserted HI or asserted LO). A typical HOME sensor implementation is to have an optical sensor and a flag at the stepper motor shaft. When the flag meets the optical sensor slit, then the stepper motor stops and this position is from now on referred to as HOME. Internally, the controller clears the StepPosition variable.

Since the motor can be started at any given position, the HOME sensor could be found to be at either state. Hence, initially, the state is to be considered unknown. The typical homing implementation calls for a transition from HI to LO or a transition from LO to HI as chosen by the application.

In order to easily capture the chosen transition, the HOME sensor was allocated to a GPIO pin with a Pin On Change Interrupt. Since the pin can be configured to raise the ISR flag either with a rising or a falling edge, we can capture on either edge. Whereas the typical polling function would require some sort of a state machine to filter out the wrong transition, this hardware interrupt works exceptionally well, rendering the amount of code to be considerably tiny.Figure 2-11 shows the ISR for the PORT1 Pin On Change Interrupt vector. The reader will notice this code is in charge of clearing the StepPosition variable, as well as stopping the motor.

```
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_Change(void)
{
  if (P1IFG && HOMEIN)
  {
    P1IE = 0;
    P1IFG = 0;
    TA0CCTL2 &= ~(CCIE + BIT5 + BIT6 + BIT7);
    StepPosition = 0;
  }
}
```

Figure 2-11. Port 1 Pin On Change Interrupt Service Routine

3 I²C Protocol and Communications Engine

In order to program the device parameters and the stepper motion engine profiles, an I²C protocol was chosen. The way it was designed, up to four controllers can be cascaded with only two communication lines and two address selector lines.

The I²C protocol is the typical three byte packet where the first byte is the target address, the second byte is the register address and the third byte is the data. There are 21 possible address register which can be accessed.

Index	PARAMETER TABLE
0x00	Number Of Steps
0x01	Number Of Steps
0x02	Number Of Steps1
0x03	Number Of Steps0
0x04	StepsToStop
0x05	StepsToStop
0x06	StepsToStop1
0x07	StepsToStop0
0x08	StartSpeed1
0x09	StartSpeed0
0x0A	Accel1
0x0B	Accel0
0x0C	DesiredSpeed1
0x0D	DesiredSpeed0
0x0E	Decel1
0x0F	Decel0
0x10	GPIO CONFIG
0x11	Stepper Config
0x12	GPIO OUT
0x13	Current Duty Cycle
0x14	Start Stepper

Figure 3-1. Parameters Table

The variables residing on addresses 0x00 to 0x0F have been discussed on previous sections. The other addresses are a combination of parameters as well as actions.

3.1 GPIO CONFIG

Defines the GPIO direction, for the 8 GPIO pins 0 to 7, where a configuration of 0 denotes an input and a configuration of 1 denotes an output.

Bit 7							Bit 0
GPIO DIR 7	GPIO DIR 6	GPIO DIR 5	GPIO DIR 4	GPIO DIR 3	GPIO DIR 2	GPIO DIR 1	GPIO DIR 0

3.2 STEPPER CONFIG

Configures control signals for the stepper power stage. These register bits are directly mapped to the Power Stage hardware pins, so changing the state of any of these bits immediately changes the respective pin at the power stage input.

Bit 7							Bit 0
N/A	ENABLE	MODE2	MODE1	MODE0	N/A	N/A	DIR

3.3 GPIO OUT

Those GPIO bits which were configured as outputs are configured when writing to this address. Note that writing to this address is equivalent to writing to the MSP430 PxOUT register so only bits which are configured to be outputs on their respective PxDIR register, will react accordingly. Those pins which are configured as inputs will still behave as inputs.

Bit 7							Bit 0
GPIO 7	GPIO 6	GPIO 5	GPIO 4	GPIO 3	GPIO 2	GPIO 1	GPIO 0

3.4 Current Duty Cycle

A register accepting a number from 0 to 249 which then becomes PWM output duty cycle. The current code does not check whether the written is equal or less than 249. Any number larger than 249 will yield a 100% duty cycle PWM.

3.5 START STEPPER

This address is more a command than an actual register. A switch case statement decodes the data byte and an action is executed. Possible actions are:

OPCODE	START STEPPER
0x00	START ACCEL
0x01	START STEPPER
0x02	STOP STEPPER
0x03	CHANGE SPEED
0x04	HOME HI
0x05	HOME LO

Where:

START ACCEL starts the stepper motor through at the StartSpeed and ramps up until reaching DesiredSpeed. If no other command is received, motor stops once the NumberOfSteps have been executed.

START STEPPER starts the stepper at the StartSpeed. No acceleration occurs. If no other command is received, motor stops once the NumberOfSteps have been executed.

STOP STEPPER stops the stepper while ramping down through the deceleration profile. Motor stops as soon as the StartSpeed is reached or the NumberOfSteps have been executed.

CHANGE SPEED accelerates or decelerates the stepper depending on whether the actual speed is larger or lesser than the new DesiredSpeed. For this command to work, a new DesiredSpeed must be written while the motor is running.

HOME HI starts the motor at the StartSpeed and runs until a transition to HI is observed on the HOME sensor input, or when the NumberOfSteps have been executed.

HOME LO starts the motor at the StartSpeed and runs until a transition to LO is observed on the HOME sensor input, or when the NumberOfSteps have been executed.

The Start Stepper command will configure the stepping engine parameters such as NumberOfSteps, StepsToStop, StartStepperSpeed, DesiredStepperSpeed, AccelerationRate and DecelerationRate. It will then decode the action parameter and start the stepper motor according to the received command. The I²C communications are handled by the USCI RX and TX vector interrupts.

```
#pragma vector=USCIAB0TX_VECTOR //UCA_TRANSMIT on UART/SPI;
UCB_RECEIVE, UCB_TRANSMIT on I2C
__interrupt void USCI_AB0_Transmit(void)
{
 if (UCB0CTL1 & UCTR)
 {
  UCB0CTL1 &= ~UCTR;
  IFG2 &= ~UCB0TXIFG;
 }
 else
 {
 SerialBuffer[SerialPointer] = UCB0RXBUF;
 SerialPointer += 1;
 if (SerialPointer == SERIAL_BUFFER_LENGTH)
  {
  SerialPointer = 0;
  ParametersTable[ADDRESS] = PARAMETER;

  switch(ADDRESS)
   {
   case GPIO_CONFIG:
    char tempOut;
    P1DIR = PARAMETER & 0xC0;                    //Use 2 MSB's to configure the
GPIO Direction on pins P1.7 and P1.6
    P2DIR = PARAMETER & 0x3F;                    //Use 6 LSB's to configure the
GPIO direction on pins P2.0 to P2.5
   case STEPPER_CONFIG_ADDR:
    tempOut = P3OUT;
    tempOut &= ~(nENABLE + MODE0 + MODE1 + MODE2 + DIR);
    tempOut |= PARAMETER;
    P3OUT = tempOut;       break;
   case GPIO_OUT_ADDR:
    P2OUT = PARAMETER;
    tempOut = P1OUT;
    tempOut &= ~(BIT7 + BIT6);
    tempOut |= (PARAMETER & 0xC0);
    P1OUT = tempOut;
    break;
   case CURRENT_DC_ADDR:
    TA1CCR1 = PARAMETER;
    break;
```

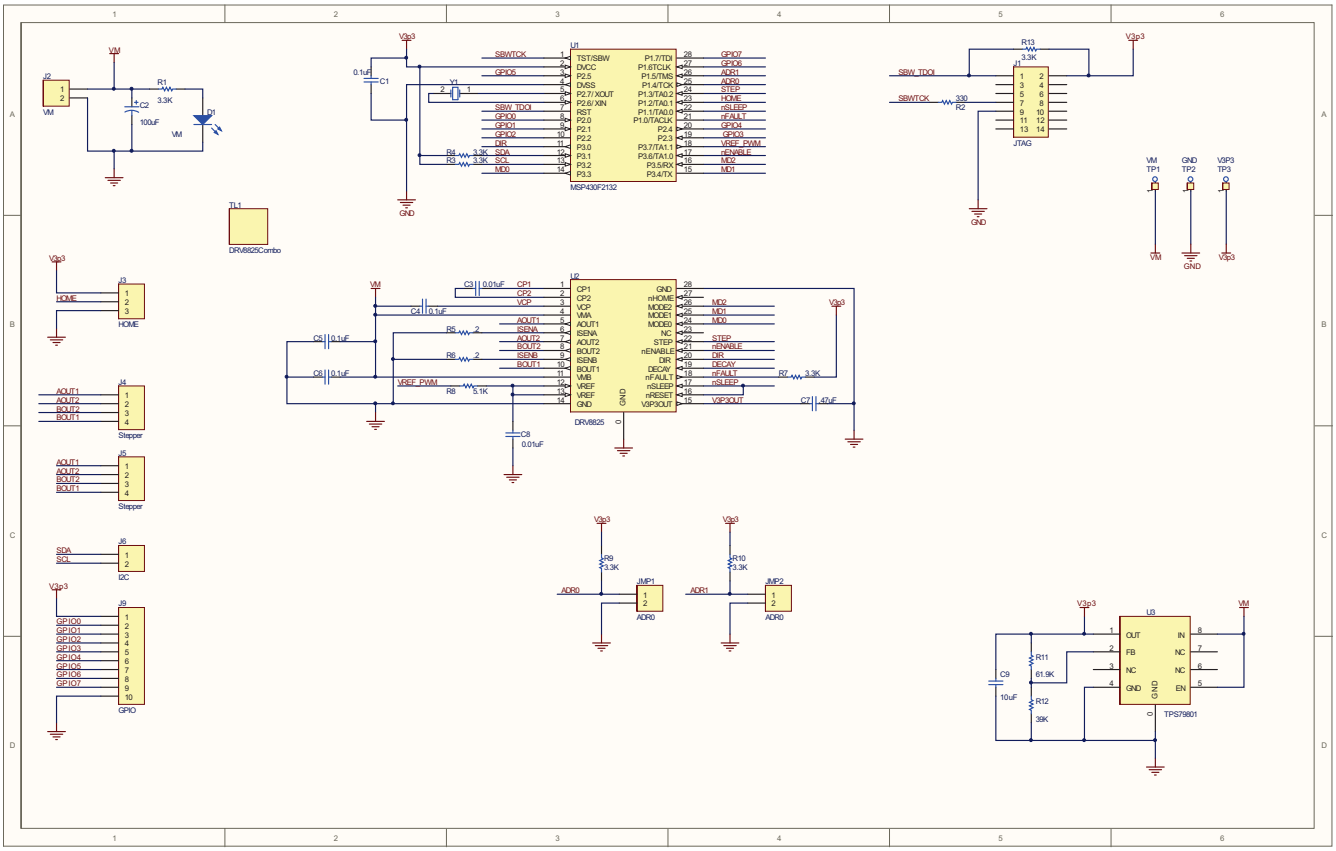
```
    case START_STEPPER_ADDR:
      NumberOfSteps = (ParametersTable[NUMBER_OF_STEPS3_ADDR] << 8) +
(ParametersTable[NUMBER_OF_STEPS2_ADDR]);
      NumberOfSteps *= 65536;
      NumberOfSteps +=  (ParametersTable[NUMBER_OF_STEPS1_ADDR] << 8) +
ParametersTable[NUMBER_OF_STEPS0_ADDR];
      ExecutedSteps = 0;

      StepsToStop = (ParametersTable[STEPS_TO_STOP3_ADDR] << 8) +
(ParametersTable[STEPS_TO_STOP2_ADDR]);
      StepsToStop *= 65536;
      StepsToStop +=  (ParametersTable[STEPS_TO_STOP1_ADDR] << 8) +
ParametersTable[STEPS_TO_STOP0_ADDR];

      StartStepperSpeed = (ParametersTable[START_SPEED1_ADDR] << 8) +
ParametersTable[START_SPEED0_ADDR];

      DesiredStepperSpeed = (ParametersTable[DESIRED_SPEED1_ADDR] << 8) +
ParametersTable[DESIRED_SPEED0_ADDR];

      AccelerationRate = (ParametersTable[ACCEL1_ADDR] << 8) +
ParametersTable[ACCEL0_ADDR];

      DecelerationRate = (ParametersTable[DECEL1_ADDR] << 8) +
ParametersTable[DECEL0_ADDR];
```

```
switch (PARAMETER)
        {
        case START_ACCEL:        // Start stepper motor at Start speed and until
reaching desired speed
          AccelerationState = ACCEL;
          ActualStepperSpeed = StartStepperSpeed;
          AccelTimeCompute(AccelerationRate);
          TA1CCTL0 |= CCIE;       //ENABLE 250 us coordinator interrupt on TA1.0
          //WDTCTL = 0x5A00 + WDTTMSEL + WDTSSEL + BIT1 + BIT0;
          break;
        case START_STEPPER:      // Start stepper motor at start speed. Run at this
rate.
          AccelerationState = NOACC;
          ActualStepperSpeed = StartStepperSpeed;
          break;
        case STOP_STEPPER:       // Stop motor from current speed, through a
deceleration profile and until reaching Start Speed. Then stop and disable stepping
engine
          AccelTimeCompute(DecelerationRate);
          AccelerationState = STOP;
          TA1CCTL0 |= CCIE;       //ENABLE 250 us coordinator interrupt on TA1.0
          break;
        case CHANGE_SPEED:       // Modify current speed up or down to Desired
Speed.
          if (DesiredStepperSpeed >= ActualStepperSpeed)
           {
           AccelTimeCompute(AccelerationRate);
           AccelerationState = ACCEL;
           TA1CCTL0 |= CCIE;       //ENABLE 250 us coordinator interrupt on TA1.0
           }
          else
           {
           AccelTimeCompute(DecelerationRate);
           AccelerationState = DECEL;
           TA1CCTL0 |= CCIE;       //ENABLE 250 us coordinator interrupt on TA1.0
           //WDTCTL = 0x5A00 + WDTTMSEL + WDTSSEL + BIT1 + BIT0;
           }
          break;
```

```
          case HOME_HI:           // Move Stepper at Start Speed until the Home
Sensor becomes HI. Go through LO if already at HI
            AccelerationState = NOACC;
            ActualStepperSpeed = StartStepperSpeed;
            P1IFG &= ~HOMEIN;
            P1IES |= HOMEIN;
            P1IE |= HOMEIN;
            break;
          case HOME_LO:           // Move Stepper at Start Speed until the Home
Sensor becomes LO. Go through HI if already at LO
            AccelerationState = NOACC;
            ActualStepperSpeed = StartStepperSpeed;
            P1IFG &= ~HOMEIN;
            P1IES &= ~HOMEIN;
            P1IE |= HOMEIN;
            break;
          }

        if (SpeedCompute(ActualStepperSpeed))
          {
          P3OUT &= ~(nENABLE);
          P1OUT |= NSLEEP;

          TA0CCR2 += StepperSpeedTMR;
          TA0CCR0 = TACCR2 + StepPulseWidth;          //2 us at 16 MHz
          TA0CCTL2 &= ~CCIFG;
          TA0CCTL2 |= (CCIE + TA0_OUTMOD2_CONF);

          }
        break;

      }
    }
  }
}
```

4 Application Schematic

The following page contains the MSP430 and DRV8811/18/24/25 combo board schematic. The stepper driver in the schematic can be replaced by DRV8434A or DRV8434/36/24/25/26/28. External current sense resistors will not be needed for these devices.



5 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision A (January 2014) to Revision B (January 2021)

Replace DRV8811/18/24/25 with DRV8434/28/26/24 in whole documents.

Changes from Revision A (January 2014) to Revision B (January 2021)	Page
• Added Integrated Stepper Motor Drivers table	1

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (https:www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265