National Semiconductor

National Semiconductor
Application Note ???
Armin Stingl

Micro-Stepper Motor Controller using the Versatile-Timer-Unit

1.0 Introduction

The CR16MCS9 microcontroller includes, among many other features, a Versatile Timer Unit (VTU). This VTU contains four fully independent 16-bit timer subsystems. It can operate as Input Compare unit or can generate a up to eight PWM signals with configrable period and duty cycle. Two of the four VTU subsystems can be combined to control one bipolar micro-stepper motor. This application note shows how the features of the VTU can be utilized to implement a stepper motor controller in a very efficient way.
The first section of this paper describes the functionality of the VTU when it operates in “Dual 8-bit PWM Mode” including a patented feature to manage a software independent update of the PWM period and duty cycle settings.
The second section provides some basics on micro stepper motors and describes the concepts of a stepper motor controller.
In the last section an example is given to illustrate in details how a micro stepper controller implementation can be derived from these concepts by utilizing the features of the Versatile Timer Unit.

The micro-stepper motor controller using the VTU of the CR16MCS9 has the following features:

- Support of up to 2 bipolar micro-stepper motors (8 micro-steps)
- Outputs can drive up to 1.6 mA per coil
  (external drivers can be used to increase the maximum coil current)
- No feedback path to sense the coil current
- PWM output with 8-bit resolution
- A nine-state state-machine controls the micro-stepping sequence.
- PWM duty-cycle configuration implemented as look-up table

The source code, presented in this paper, requires the usage of the IAR Systems toolset for the Compact RISC CR16.

© 2000 National Semiconductor Corporation

2.0 The Versatile-Timer-Unit

The Versatile Timer Unit (VTU) is comprised of four timer subsystems. Each timer subsystem contains an 8-bit clock prescaler, a 16-bit up-counter and two 16-bit registers. Each timer subsystem controls two I/O pins which either function as PWM outputs or capture inputs depending on the mode of operation. There are four system level interrupt requests, one for each timer subsystem. All four timer subsystems are fully independent and each may operate as a dual 8-bit PWM timer, a 16-bit PWM timer or as a dual 16-bit capture timer. Figure 1 illustrates the main elements of the Versatile Timer Unit (VTU).

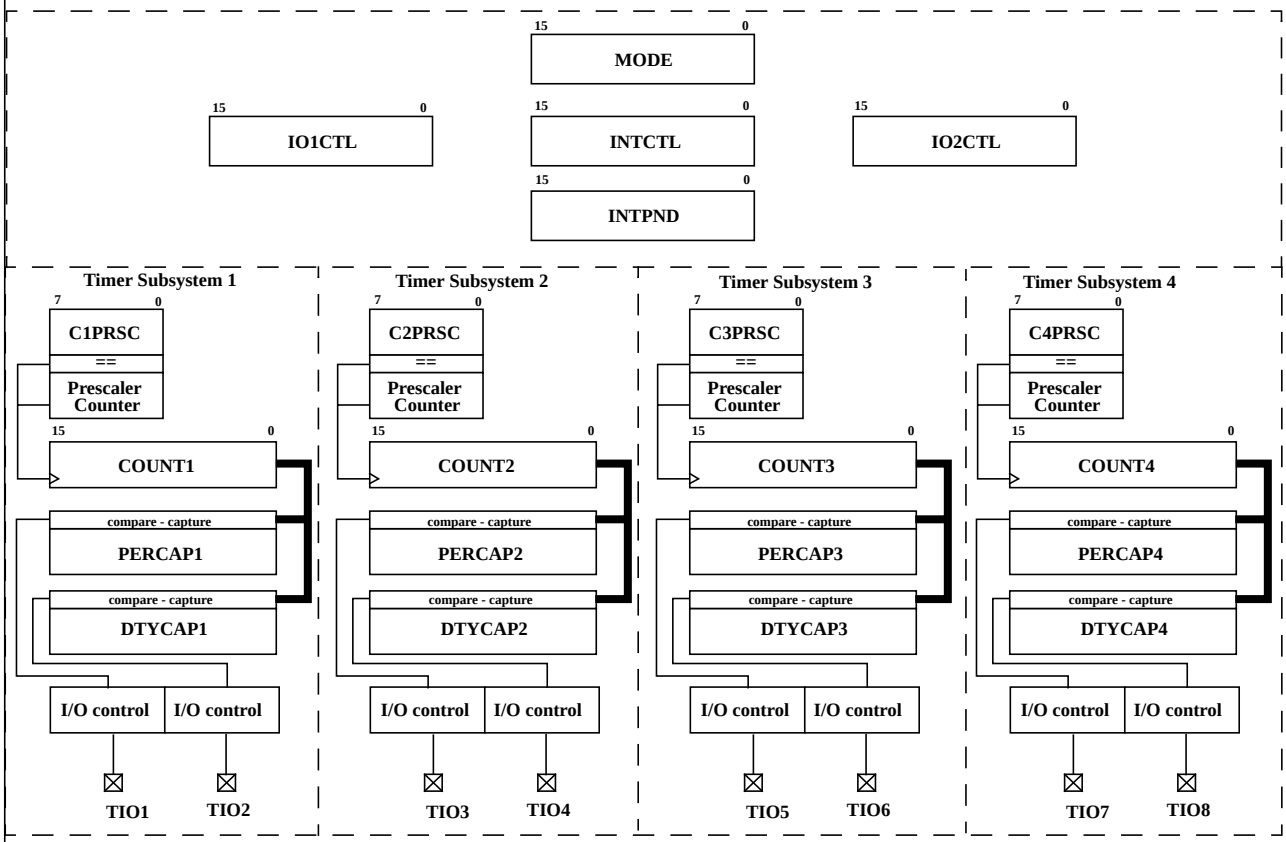

Figure 1. VTU Blockdiagram

2.1 Dual 8-bit PWM Mode

For the implementation of the stepper motor controller the VTU operates in the dual 8-bit mode. The following section describes the functionality of the VTU in this mode. For a detailed description of all other operating mode, please refer to the user manual of the CR16MCS9.

Each timer subsystem may be configured to generate two fully independent PWM waveforms on the respective TIOx pins. In this mode, the counter COUNTx is split and operates as two independent 8-bit counters. Each counter increments at the rate determined by the clock prescaler.

Each of the two 8-bit counters may be started and stopped separately via the associated TxRUN bits. Once either of the two 8-bit timers is running the clock prescaler starts counting. Once the clock prescaler counter value matches the value of the associated CxPRSC register field, COUNTx is incremented.

The period of the PWM output waveform is determined by the value of the PERCAPx register. The TIOx output starts at the default value as programmed via the IOxCTL.PxPOL bit. Once the counter value reaches the value of the period register PERCAPx, the

© 2000 National Semiconductor Corporation

counter is reset to 00_{16} upon the next counter increment. Upon the following increment from 00_{16} to 01_{16}, the TIOx output will change to the opposite of the default value.

The duty cycle of the PWM output waveform is controlled by the DTYCAPx register value. Once the counter value reaches the value of the duty cycle register DTYCAPx, the PWM output TIOx changes back to its default value upon the next counter increment. Figure 2 illustrates this concept.

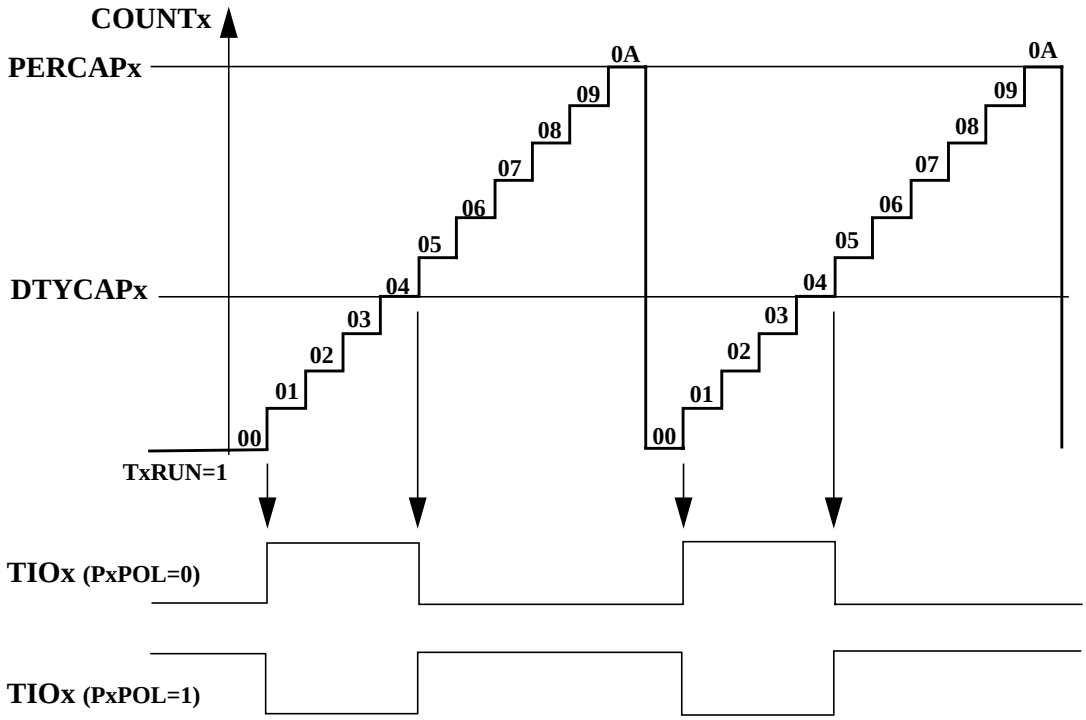

Figure 2. PWM Signal Generation with the VTU

The period time is determined by the following equation:

$$PWMperiod = (PERCAPx + 1) * (CxPRSC + 1) * T_{CLK}$$

The duty cycle in percent is calculated as follows:

$$DutyCycle[\%] = (DTYCAPx / (PERCAPx+1)) *100$$

If the duty cycle register (DTYCAPx) holds a value which is greater then the value held in the period register (PERCAPx) the TIOx output will remain at the opposite of its default value which corresponds to a duty cycle of 100%. If the duty cycle register (DTYCAPx) register holds a value of 00_{16}, the TIOx output will remain at the default value which corresponds to a duty cycle of 0%. In that case the value contained in the PERCAPx register is irrelevant. This scheme allows the duty cycle to be programmed in a range from 0% to 100%.

In order to allow fully synchronized updates of the period and duty cycle compare values, the VTU uses a patented update sequence. The PERCAPx and DTYCAPx registers are double buffered when operating in PWM mode. Therefore, if the user writes to either the period or duty cycle register while either of the two PWM channels is enabled, the new value will not take effect until the counter value matches the previous period value or the timer is stopped. In order to avoid the case in which the user software update occurs exactly at a period boundary resulting in a mix between an old period or duty cycle value and a new duty cycle or period value for one PWM period, a specific write sequence is performed by the VTU. This feature of the VTU allows the user software to update the

© 2000 National Semiconductor Corporation

period and duty cycle registers at any time while the counter is running without the jepardy that changing the duty cycle or period results in an undesired PWM waveform for the transition between the old and the new waveform. The new values will only affect the output waveform when the current period ends or when the counter is stopped. Reading the PERCAPx or DTYCAPx register will always return the most recent value written to it.

The counter registers can be written if both 8-bit counters are stopped. This allows the user to preset the counters before starting and therefore generate PWM output waveforms with a phase shift relative to one another. If the counter is written with a value other than 00_{16} it will start incrementing from that value while TIOx remains at its default value until the first 00_{16} to 01_{16} transition of the counter value occurs. If the counter is preset to values which are smaller or equal then the value held in the period register (PERCAPx) the counter will count up until a match between the counter value and the PERCAPx register value occurs. The counter will then be reset to 00_{16} and continue counting up. Alternatively the counter may be written with a value which is greater then the value held in the period register. In that case the counter will count up to FF_{16} and then roll over to 00_{16}. In any case the TIOx pin always changes its state at the 00_{16} to 01_{16} transition of the counter.

The user software may only write to the COUNTx register if both TxRUN bits of a timer subsystem are cleared. Any writes to the counter register while either timer is running will be ignored.

The two I/O pins associated with a timer subsystem function as independent PWM outputs in the dual 8-bit PWM mode. If a PWM timer is stopped via its associated MODE.TxRUN bit the following actions result:

— the associated TIOx pin will return to its default value as defined by the IOxCTL.PxPOL bit,

— the counter will stop and will retain its last value,

— any pending updates of the PERCAPx and DTYCAPx register will be completed,

— the prescaler counter will be stopped and reset if both MODE.TxRUN bits are cleared.

Figure 3 illustrates the configuration of a timer subsystem while operating in dual 8-bit PWM mode. The numbering refers to timer subsystem 1 but equally applies to the other three timer subsystems.

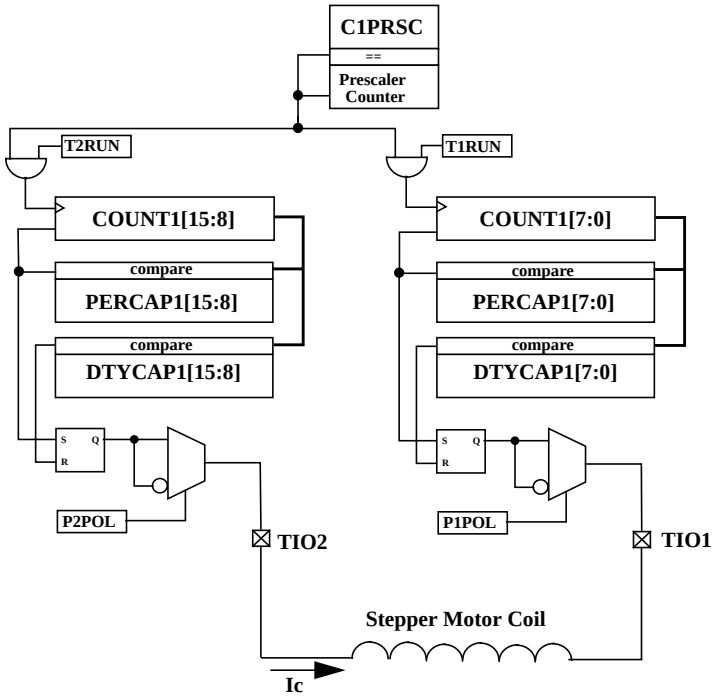

Figure 3. VTU Sub-Module 1 in Dual 8-bit PWM Mode connected to one Stepper Motor Coil

© 2000 National Semiconductor Corporation

3.0 Stepper Motor Controller Concept

The stepper motor controller implementation presented in this application note supports bipolar micro-stepping motors, which can perform 8 micro-steps in between each of the four main steps. A simplified block diagram of a bipolar stepper motor is given in figure 4. The four main steps can be accomplished by driving a constant current of a certain direction through either one of the two coils of a bipolar stepper motor. Only one coil is energized at a time.

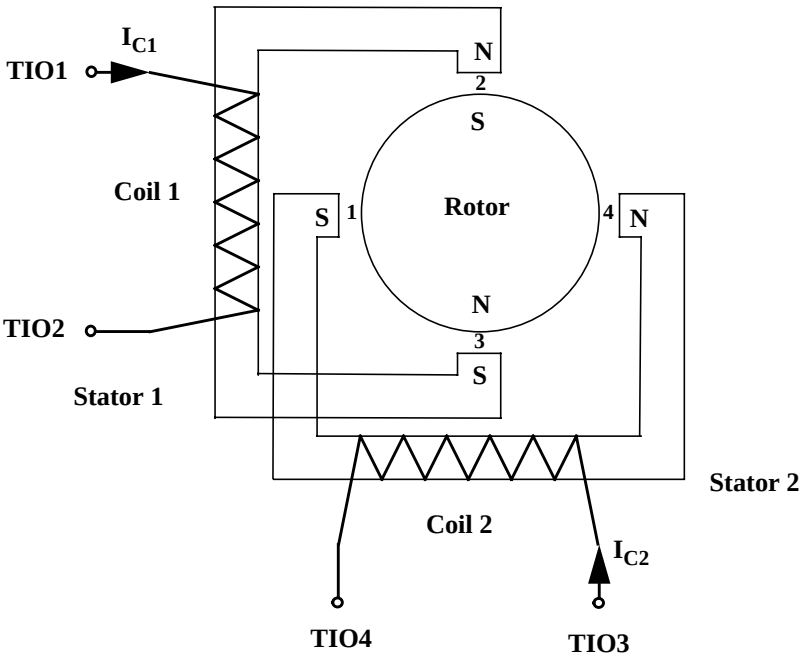

Figure 4. Simplified Bipolar Stepper Motor

The four current states (main steps) are given in table 1.

Table 1. Coil Current for four main steps

Main Step	Rotation Angle	Current through Coil 1	Current through Coil 2
1	0^o	0%	-100%
2	90^o	+100%	0%
3	180^o	0%	+100%
4	270^o	-100%	0%

In order to achieve a higher stepping resolution (smaller stepping angle) so-called micro steps are inserted in between the four main steps. These micro steps can be accomplished by driving both coils of the motor with a reduced current level at the same time.
The most common approach to control the current through the coils is to apply a pulsed signal to the coils rather than a constant voltage. The inductance of the motor coils integrates the voltage pulses to an almost constant current level. The current level can be adjusted by simply controlling the duty cycle of the pulsed signal. In other words, a defined coil current is accomplished by a pulse width modulated (PWM) voltage across the stepper motor coil. Figure 5 shows the simplified relationship between the PWM signal and the current through the coil. In practice the current level will not be exactly constant since the inductance of the coil is not ideal.

© 2000 National Semiconductor Corporation

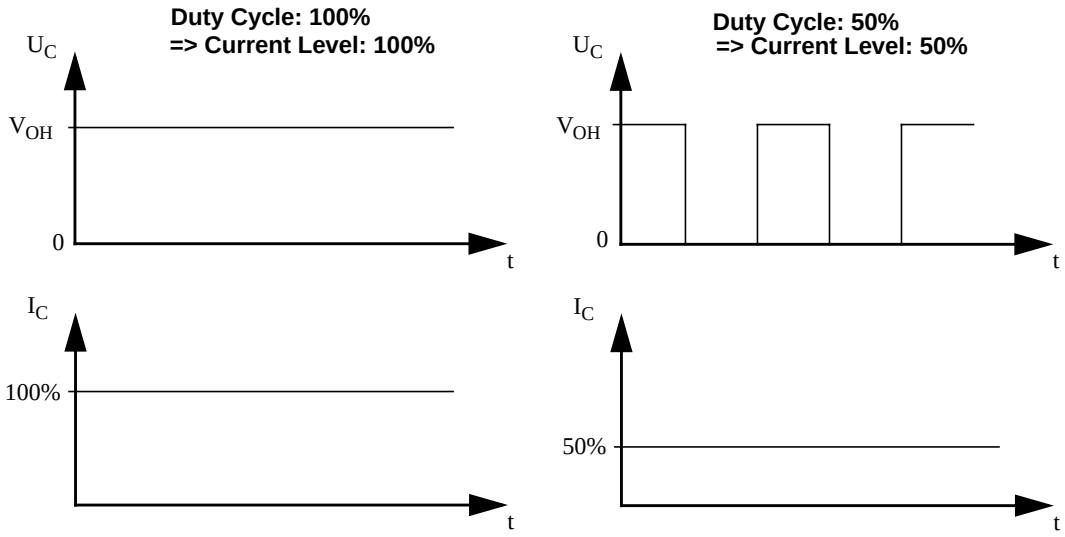

Figure 5. PWM duty cycle and Coil Current Level

As shown in figure 6 each main step (90^o) is subdivided into 8 equal micro-steps of 11.25^o. It also shows that the current levels through each coil can take one out of 9 absolute values (0%, 22%, 42%, 57.5%, 72.5% 84%, 91% and 100%). These nine absolute values correspond to nine different PWM duty cycle settings. Furthermore, the whole rotation can be divided into four sectors, representing the four main steps. Each sector covers 90^o.

As you will see later on in the implementation of the stepper motor controller, each PWM setting corresponds to one state of a state-machine which controls the PWM configuration for each coil (I_{C1}-states and I_{C2}-states). Therefore figure 6 also gives the state-machine states for each micro-step.

The sequence of the states is unique for each sector. The first 90^o sector, where the I_{C1}-states follow the sequence 0,1,2,...8, can be called sector A. Within the second sector, sector B, the I_{C1}-states follow the sequence 8,7,...0. The third sector is identical to sector A, except that the signs for all current levels (direction) are inverted. Therefore this sector is called -A. Also the sequence of sector four is the same as for sector B, but with inverted current levels. Thus, this sector is called sector -B.

Figure 6 shows the curent levels and state-machine states when a stepper motor performs the micro-steps for one 360^o rotation.

© 2000 National Semiconductor Corporation

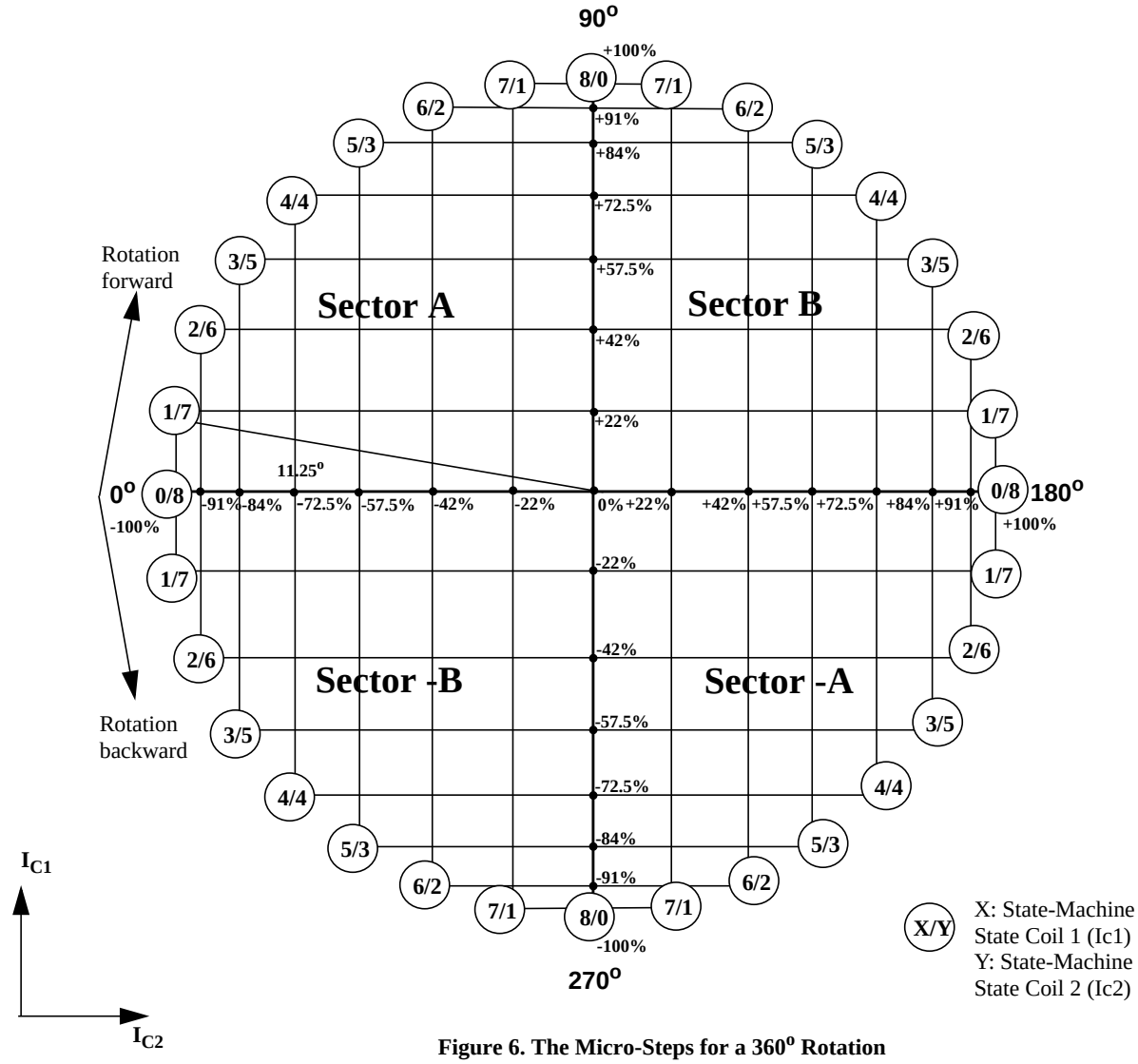

Figure 6. The Micro-Steps for a 360° Rotation

Another approach to illustrate the relationship between micro-steps, coil current and state-machine states is given in figure 7. It shows that the sequence of current levels through coil 1 of the motor builds a sin(x) function, whereas the sequence of current levels through coil 2 follows a -cos(x) function.

Due to this relationship between I_{C1} and I_{C2} there is no need to implement two separate state-machines. The current levels for I_{C2} can be derived from the state-machine for I_{C1}. Due to the 90^o phase shift between the I_{C1}-states and the I_{C2}-states, all I_{C2}-states are eight minus I_{C1}-states.

© 2000 National Semiconductor Corporation

Figure 7 shows the various coil current levels while a stepper motor with eight micro-steps performs one rotation.

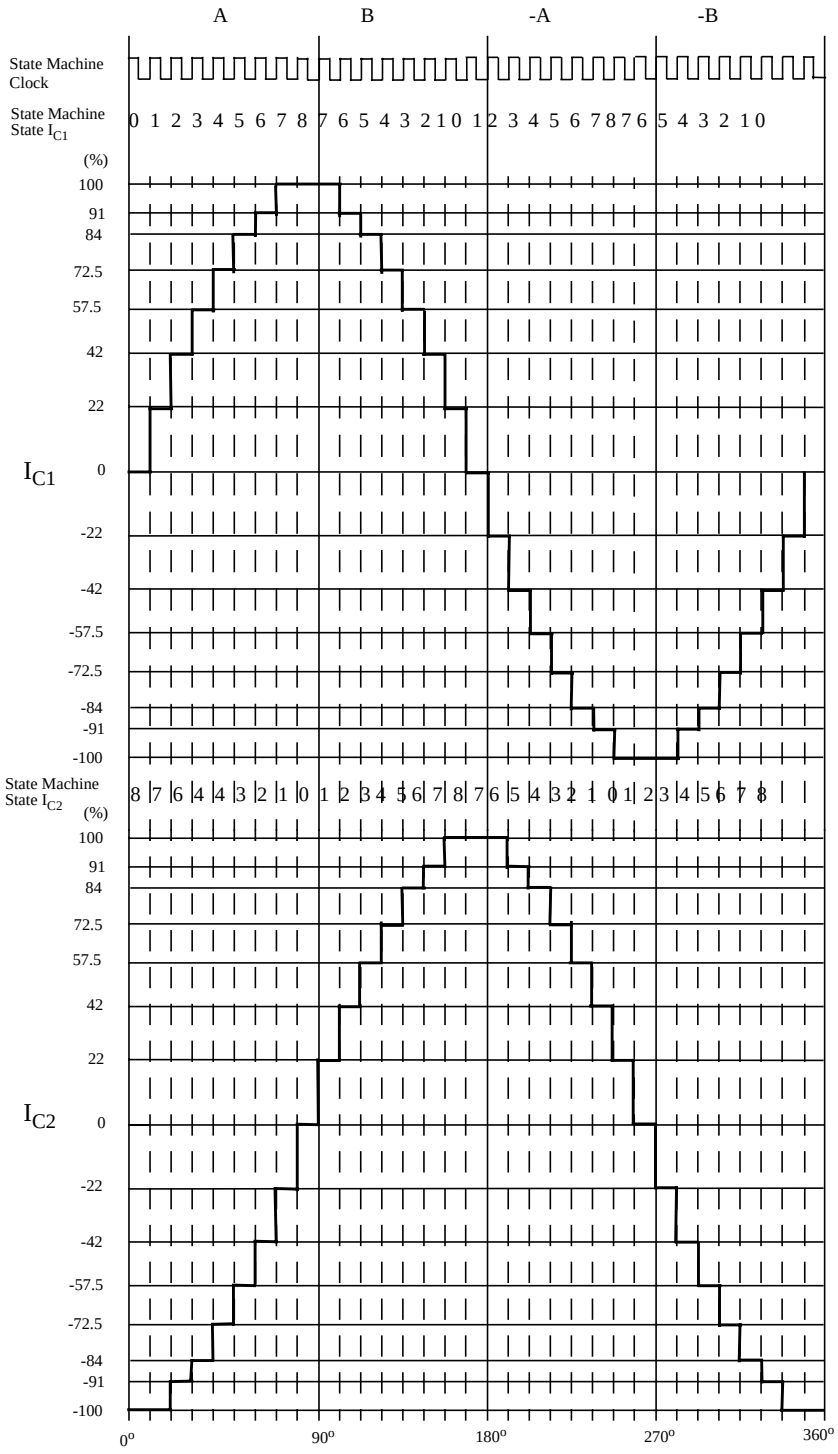

Figure 7. Coil Current Levels depending on State-Machine State (one rotation)

© 2000 National Semiconductor Corporation

4.0 Implementation

The Versatile Timer Unit (VTU) of the CR16MCS9 controls the current through the stepper motor coils by means of a PWM signal with configurable duty cycle. As this implementation supports stepper motors with up to eight micro-steps, the controller is able to generate up to 9 pre-configurable coil current levels, hence 9 different PWM duty cycles. The PWM duty cycles can be pre-defined and adjusted to the individual requirements by means of a look-up table residing in the flash program memory. This approach uses the two 8-bit PWM channels of each of the four VTU sub-modules to control one stepper motor coil. Both 8-bit PWM channels of one subsystem are configured to produce the same PWM signal. However, one of the two PWM output pins is set to constant low-level (GND), whereas the other pin actually generates the PWM signal. Switching the constant low-level and the PWM signal between the two output pins controls the direction (+/-) of the current through the coil. The 8-bit counter, COUNT1 and the PERCAP register are configured to produce a PMW with constant period. The value of the PWM period mainly depends on the main system clock, used within the individual application, but should be above the audible frequency range, i.e. it should be above 20kHz. If the PWM periode is within the audible frequency range, the motor coils may oscillate at this frequency. This may produce a buzzing sound. The requirement of a minimum PWM frequency of 20kHz results in a minimum system clock of 256 x 20kHz = 5.12MHz. A 9-state state-machine coordintes the configuration of the duty cycle register, DTYCAP, depending on the rotation direction and the desired number of micro-steps (rotation angle). The state-machine loads the appropriate entry of the look-up table into the duty cycle registers of the VTU. The state-machine is controlled by the idle timer T0. Triggered by a T0 interrupt (T0 underflow) the state (PWM duty cycle) is changed according to the rotation direction. This means the rotation speed is defined by the frequency of T0 underflows, hence, by the counter T0 and its prescaler settings.

4.1 Current Level/Duty Cycle Settings

As already mentioned, both PWM channels of each VTU subsystem generate the same PWM signal. Since the 8-bit period and duty cycle configuration registers for both 8-bit PWM channels are combined to form 16-bit registers, both channels can be set simultaniously by simply writing 16-bit values to the PERCAPn and DTYCAPn registers. In order to utilize the full 8-bit PWM resolution of the VTU the period register, PERCAPn, is set to its maximum value 0xFFFF (0xFF for both channels). The various DTYCAPn values, stored in a look-up table, to achieve the required current levels are given in table 2.

Table 2. DTYCAPn values stored in the look-up table

State-Machine State	Coil Current Levels in %	DTYCAPn value
0	0	0x0000
1	22	0x3838
2	42	0x6B6B
3	57.5	0x9393
4	72.5	0xBEBE
5	84	0xD6D6
6	91	0xE8E8
7	100	0xFFFF
8	100	0xFFFF

Due to the 8-bit resolution the coil current levels can be set with an accuracy of +/- 0.4%. Further adjustments of the DTYCAPn values to the individual motor may be required by measuring teh various current levels.
The C-code below shows a possible implementation of the look-up table as discribed in table 2.

```
const unsigned int C_DTY_STATE[9] =
{
   // DTYCAP    State     IC1   IC2
   //*******************************
   0x0000,   // state 0:   0.0, 100.0
   0x3838,   // state 1:  22.0, 100.0
   0x6B6B,   // state 2:  42.0,  91.0
```

© 2000 National Semiconductor Corporation

```
   0x9393,   // state 3:  57.5,  84.0
   0xBEBE,   // state 4:  72.5,  72.5
   0xD6D6,   // state 5:  84.0,  57.5
   0xE8E8,   // state 6:  91.0,  42.0
   0xFFFF,   // state 7: 100.0,  22.0
   0xFFFF,   // state 8: 100.0,   0.0
};
```

As already mentioned there is a simple relationship between the two states of the state machines for the two coil currents I_{C1} and I_{C2}. The state for I_{C2} is always equal to eight minus the state of I_{C1}. Therefore, only one state-machine needs to be implemented, which can control both coil currents and also the look-up table only needs to be stored once. In other words, only the state-machine for coil 1 must be implemented. The states to control the current through coil 2 can be derived from the states of coil 1.
The state-machine needs to take into account, in which sector (A, B, -A or -B) it is currently operating and also the desired rotation direction of the motor, in order to distinguish, whether to increment or to decrement the state number. The state-machine also controls the PWM output pins depending on the sector in which it is operating. Table 3 summarizes the I/O configuration for each section.

Table 3. PWM terminal configuration

Sector	Sector Number	TIO1	TIO2	TIO3	TIO4
A	0	PWM	GND	GND	PWM
B	1	PWM	GND	PWM	GND
-A	2	GND	PWM	PWM	GND
-B	3	GND	PWM	GND	PWM

In the proposed implementation the PWM terminal configuration is also stored in a look-up table. The fact, that all terminals use four pins of the same 8-pin port (Port F) simplifies the data structure of the look-up table. The implementation of the look-up table is shown in the C-code below. The data byte of the array is loaded into the alternate function register of port F (PFALT). A bit set to 1 means, that the alternate function is enabled and the pin is used as the PWM channel. If a bit is cleared to 0, the pin is used as general purpose I/O and driven to a logic-low level. The register PFDIR is permanently set to 0xFF whereas the register PFDOUT is set to 0x00.

```
const unsigned char C_IO_STATE[4] =
{
   // PFALT  TIO4   TIO    TIO2   TIO1
   //        PF7    PF6    PF3    PF2
   0x84,  //  PWM    GND    GND    PWM
   0x44,  //  GND    PWM    GND    PWM
   0x48,  //  GND    PWM    PWM    GND
   0x88   //  PWM    GND    PWM    GND
};
```

4.2 The State-Machine

The state-machine is triggered by a T0 interrupt. This means, the state changes everytime the timer T0 underflows. The number of state changes is specified by the desired number of micro-steps, in other words, by the desired rotation angle. The timer T0 is configured to generate interrupt requests on a regular basis. The frequency of T0 interrupts defines the rotation speed of the stepper motor. When the motor is supposed to start moving, the number of micro-steps is loaded into a globaly defined counter variable and the T0 interrupts are enabled. Everytime a T0 interrupt is generated, the state-machine is entered and the new PWM duty-cycle is set. The state-machine needs to perform the following tasks:

— Set the VTU and PWM terminal configuration depending on the new state.

— Set a new state depending on the current sector and the rotation direction. For state 0, the state number always needs to be incremented, for state 8 always decremented.

— Set a new sector, depending on the rotation direction. A new section only needs to be set, if the current state number is either 0 or 8.

— Decrement the micro-step counter.

— Disable further T0 interrupts when the desired number of micro-steps have been executed.

© 2000 National Semiconductor Corporation

The following C-code shows a possible implementation of the state-machine.

```
#pragma function = __interrupt
void T0_IRQ(void)
{
   volatile unsigned char dummy;

   // clear T0_IRQ pending (clear on read)
   dummy = T0CSR.byte;

   // set the VTU according to the new state, UC_SMC_STATE
   switch(UC_SMC_STATE)
   {
      //------------------------------
      case 0:   set_VTU(0, UC_SMC_SECTOR);
                set_new_sector(UC_DIRECTION);
                UC_SMC_STATE++;
                break;
      case 1:   set_VTU(1, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 2:   set_VTU(2, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 3:   set_VTU(3, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 4:   set_VTU(4, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 5:   set_VTU(5, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 6:   set_VTU(6, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 7:   set_VTU(7, UC_SMC_SECTOR);
                set_new_state(UC_DIRECTION);
                break;
      case 8:   set_VTU(8, UC_SMC_SECTOR);
                set_new_sector(UC_DIRECTION);
                UC_SMC_STATE--;
                break;
      default:;
   }

   // decrement micro-step counter
   UI_IRQ_COUNT_DOWN--;

   // disable T0 IRQ, when desired number of
   // micro-steps have been performed
   if(!UI_IRQ_COUNT_DOWN) T0CSR.bit.T0INTE = 0;
}
#pragma function = default
```

The T0 interrupt handler uses three additional functions:

— set_VTU
— set_new_state
— set_new_sector

These functions are explained in the following sections:

© 2000 National Semiconductor Corporation

4.3 The Function “set_VTU”

The function “set_VTU” sets the new VTU configuration to generate the new required PWM signal to force the motor to perform one micro-step of the desired rotation direction. As the new duty cycle settings depend on the next state and on the current sector, the two parameter “UC_SMC_STATE” and “UC_SMC_SECTOR” are passed to this function. The function gets the new I/O settings from the look-up table C_IO_STATE depending on the sector and configures the port F accordingly. It also gets the new duty cycle settings for both coil currents (both VTU subsystems) from the look-up table C_DTY_STATE depending on the next state and loads them into the DTYCAP registers. The code sequence to update the DTYCAP registers is optimized for maximum execution speed in order to keep the overlap between the old and new coil currents through both coils as short as possible.

```
void set_VTU(unsigned char uc_pwm_state, unsigned char uc_io_state)
{
   register unsigned int uir_new_dtycap1;
   register unsigned int uir_new_dtycap2;
   register unsigned int uir_new_io_set;

   // load new Port F IO setting
   // first in CR16B registers
   uir_new_io_set = C_IO_STATE[uc_io_state];

   // load new PWM duty cycle value into DTYCAP
   // first load value in CR16B registers
   uir_new_dtycap1 = C_DTY_STATE[uc_pwm_state];
   uir_new_dtycap2 = C_DTY_STATE[8-uc_pwm_state];

   // load new PWM setting into VTU register
   // (update both DTYCAP as simultaniously as possible)
   DTYCAP1 = uir_new_dtycap1;
   DTYCAP2 = uir_new_dtycap2;

   // load new IO settings into PF alternate function
   // register
   PFALT.byte = uir_new_io_set;

   // start COUNT1 and COUNT2 in dual 8-bit PWM mode
   MODE.word |= 0x0033;
}
```

4.4 The Function “set_new_state”

The function “set_new_state” calculates the next state of the state-machine depending on the rotation direction and the current sector. When the rotation direction is forward and the current sector number is either 0 or 2, then the state number is incremented; for the sector 1 and 3 the state number is decremented. Furthermore, when the rotation direction is backward and the current sector number is either 0 or 2, then the state number is decremented; for the sector 1 and 3 the state number is incremented. The function also takes care that the state number can only range between the values 0 through 31.

```
void set_new_state(unsigned char uc_direction)
{
   if(uc_direction == FORWARD)
   {
      if(UC_SMC_SECTOR & 0x01)
      {
         UC_SMC_STATE--;
         if(UC_SMC_STATE == 255) UC_SMC_STATE = 31;
      }
      else
      {
         UC_SMC_STATE++;
         if(UC_SMC_STATE == 32) UC_SMC_STATE = 0;
      }
```

© 2000 National Semiconductor Corporation

```
    }
    else
    {
        if(UC_SMC_SECTOR & 0x01)
        {
            UC_SMC_STATE++;
            if(UC_SMC_STATE == 32) UC_SMC_STATE = 0;
        }
        else
        {
            UC_SMC_STATE--;
            if(UC_SMC_STATE == 255) UC_SMC_STATE = 31;
        }
    }
}
```

4.5 The Function “set_new_sector”

The function “set_new_sector” increments the sector number depending on the rotation direction. It also takes care that the sector number can only range between 0 and 3.

```
void set_new_sector(unsigned char uc_direction)
{
    if(uc_direction == FORWARD)
    {
        UC_SMC_SECTOR++;
        if(UC_SMC_SECTOR == 4) UC_SMC_SECTOR = 0;
    }
    else
    {
        UC_SMC_SECTOR--;
        if(UC_SMC_SECTOR == 255) UC_SMC_SECTOR = 3;
    }
}
```

© 2000 National Semiconductor Corporation

5.0 Code Listing

This section lists the entire code for a sample stepper motor controller implementation. The source code is written in C for the CompactRISC CR16 Embedded Workbench from IAR Systems. The entire project consists of two modules:

- SMC_VTU.C
- SMC_VTUS.C

Furthermore, the XLINK linker of the IAR Systems toolset requires a definition file to specify the memory allocation, depending on the memory map of the device and the application requirements. This linker definition file, CR16MCS9.XCL, is also described in one of the following sections.
The C start-up routine, CSTARTUP.S41, shows how to set-up the CR16B core to support High-Level language programming.

5.1 The Module SMC_VTU.C

This module includes the void main(void) function as well as all other functions needed for the stepper motor controller.

```
/*****************************************************************/
/*  Project:     SMC_VTU                                         */
/*  Module:      SMC_VTU.C                                       */
/*                                                               */
/*  Author:      Armin Stingl                                    */
/*               1/10/2000                                       */
/*               National Semiconductor, Germany                 */
/*                                                               */
/*   Toolset:    IAR Systems, Embedded Workbench for CR16        */
/*               Version: 1.30A                                  */
/*****************************************************************/

//*************************************************
// Include Files
//*************************************************
#include <cr16mcs9.h>
#include <incr16.h>


//*************************************************
// Defines
//*************************************************
#define FORWARD  1
#define BACKWARD 0

//*************************************************
// Constants
//*************************************************

// Look-up Table for Duty Cycle State
const unsigned int C_DTY_STATE[9] =
{
   // DTYCAP     State     IC1    IC2
   //*********************************
   0x0000,   // state 0:   0.0, 100.0
   0x3838,   // state 1:  22.0, 100.0
   0x6B6B,   // state 2:  42.0,  91.0
   0x9393,   // state 3:  57.5,  84.0
   0xBEBE,   // state 4:  72.5,  72.5
   0xD6D6,   // state 5:  84.0,  57.5
   0xE8E8,   // state 6:  91.0,  42.0
   0xFFFF,   // state 7: 100.0,  22.0
   0xFFFF,   // state 8: 100.0,   0.0
};


// Look-up for I/O Terminal Setting
const unsigned char C_IO_STATE[4] =
```

© 2000 National Semiconductor Corporation

```
{
   // PFALT  TIO4   TIO    TIO2   TIO1
   //        PF7    PF6    PF3    PF2
   0x84,  // PWM    GND    GND    PWM
   0x44,  // GND    PWM    GND    PWM
   0x48,  // GND    PWM    PWM    GND
   0x88   // PWM    GND    PWM    GND
};


//*********************************************
// Function Prototypes
//*********************************************
void BIUinit(void);
void VTUinit(void);
void TWMinit(void);
void set_new_state(unsigned char);
void set_new_sector(unsigned char);
void set_VTU(unsigned char, unsigned char);


//*********************************************
// Global Variables
//*********************************************
unsigned char UC_SMC_STATE;
unsigned char UC_SMC_SECTOR;
unsigned char UC_DIRECTION;
unsigned int  UI_STEP_COUNTER;

//*********************************************
//    M A I N
//*********************************************
void main(void)
{
   BIUinit();                 // initialize Bus Interface Unit

   TWMinit();                 // initialize Timing and Watchdog Module

   VTUinit();                 // initialize Versatile-Timer-Unit

   IENAM0.word = 0x0000;      // initialize Interrupt Control Unit
   IENAM1.word = 0x0000;
   IENAM1.bit.RTI = 1 ;       // enable only T0 interrupt IRQ31


   UC_SMC_STATE = 0;          // initialize Stepper Motor State
   UC_SMC_SECTOR = 0;         // initialize Stepper Motor Sector
   UC_DIRECTION = FORWARD;    // set Rotation Direction to forward
   UI_STEP_COUNTER = 60000;   // set Step Counter

                              // enable Interrupts in CR16B Core
   __set_processor_register(Reg_PSR, 0x0A00);

   for(;;);                   // just wait for T0 interrupts


} // of main()



//*****************************************************************************
//  Function: BIUinit()
//   Initialize Bus Interface Unit
//*****************************************************************************
void BIUinit(void)
{
    MCFG.byte   = 0x00;    // no clocks generated on ENV-pins
    BCFG.byte   = 0x06;    // BIU set to late-write
    IOCFG.word  = 0x080;   // no additional idle cycles for I/O accesses
```

© 2000 National Semiconductor Corporation

```
    SZCFG0.word = 0x880;   // fast read enabled from program memory
    SZCFG1.word = 0x880;   // fast read enabled from ISP memory
}

//*************************************************************************
// Function: TWMinit()
//  Initialize TWM for 2000 steps/second
//   (Also any other step rate can be chosen, depending on
//   stepper motor type and system frequency.)
//*************************************************************************
void TWMinit(void)
{
   CRCTL.byte = 0x00;         // prescaled high speed clock is Slow System Clock
   PRSSC = 79;                // slow clock is 16MHz/2/(79+1) = 100kHz

   TWCP.byte = 0x00;          // set T0 prescaler to 0, Div-by 1
   TWMT0 = 49;                // T0 IRQ every 0.5ms => 2000 steps/second

   T0CSR.bit.T0INTE = 1;      // enable T0 interrupts
   T0CSR.bit.RST = 1;         // start T0

}

//*************************************************************************
// Function: VTUinit()
//  Initialize VTU
//*************************************************************************
void VTUinit(void)
{
   // VTU runs in dual 8-bit mode
   // one sub-module drives two coils of one motor

   CLK1PS.word = CLK2PS.word = 0x0101;  // prescaler for all four sub-modules is 1
                                        // 16MHz/2=8MHz = 125ns

   MODE.word = 0x0044;                  // set COUNT1 and COUNT2 is dual 8-bit PWM mode

   PERCAP1 = PERCAP2 = 0xFFFF;          // PWM period is 31,25kHz (constant)
                                        // 125ns * 256 = 32us = 31,25kHz

   DTYCAP1 = DTYCAP2 = 0x0000;          // initial PWM duty cycle set to 0%

   COUNT1 = COUNT2 = 0x0000;            // intial counter value set to 0

   IO1CTL.word = 0x0000;                // PxPOL = 0 for all TIOx (x=1,2,3,4)
                                        // set to 0 upon DTYCAP match

   PFALT.byte = 0;                      // disable VTU function of all TIOx

   PFDOUT.byte = 0;                     // set all port pins to 0

   PFDIR.byte = 0xCC;                   // PF2 (TIO1), PF3 (TIO2), PF6 (TIO3), PF7 (TIO4)
                                        // are output,
                                        // all others are input

}


//*************************************************************************
// Interrupt: T0_IRQ()
//  T0 interrupt handler, SMC State-Machine
//*************************************************************************
#pragma function = __interrupt
void T0_IRQ(void)
{
   volatile unsigned char dummy;

   // clear T0 interrupt request pending (clear-on-read from T0CSR)
   dummy = T0CSR.byte;
```

© 2000 National Semiconductor Corporation

```
    switch(UC_SMC_STATE)
    {
        //------------------------------
        case 0:   set_VTU(0, UC_SMC_SECTOR);
                  set_new_sector(UC_DIRECTION);
                  UC_SMC_STATE++;
                  break;

        case 1:   set_VTU(1, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 2:   set_VTU(2, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 3:   set_VTU(3, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 4:   set_VTU(4, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 5:   set_VTU(5, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 6:   set_VTU(6, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 7:   set_VTU(7, UC_SMC_SECTOR);
                  set_new_state(UC_DIRECTION);
                  break;

        case 8:   set_VTU(8, UC_SMC_SECTOR);
                  set_new_sector(UC_DIRECTION);
                  UC_SMC_STATE--;
                  break;

        default:;
    }


    // decremnt step counter
    UI_STEP_COUNTER--;

    // disable further T0 interrupts, when step counter has reached 0
    if(!UI_STEP_COUNTER) T0CSR.bit.T0INTE = 0;


}
#pragma function = default



//*********************************************
// Function: set_VTU()
//    Configure VTU with PWM settings
//*********************************************
void set_VTU(unsigned char uc_pwm_state, unsigned char uc_io_state)
{
    register unsigned int uir_new_dtycap1;
    register unsigned int uir_new_dtycap2;
    register unsigned int uir_new_io_set;
```

© 2000 National Semiconductor Corporation

```
    // load new Port F IO setting
    // first in CR16B registers
    uir_new_io_set = C_IO_STATE[uc_io_state];

    // load new PWM duty cycle value into DTYCAP
    // load value first in CR16B registers
    uir_new_dtycap1 = C_DTY_STATE[uc_pwm_state];
    uir_new_dtycap2 = C_DTY_STATE[8-uc_pwm_state];

    // load new PWM setting into VTU register
    // (update both DTYCAP as simultaniously as possible)
    DTYCAP1 = uir_new_dtycap1;
    DTYCAP2 = uir_new_dtycap2;

    // load new IO settings into PF alternate function
    // register
    PFALT.byte = uir_new_io_set;

    // start COUNT1 and COUNT2 is dual 8-bit PWM mode
    MODE.word |= 0x0033;


}

//*****************************************************************************
//  Function: set_new_state()
//   Calculate next state of the state-machine
//*****************************************************************************
void set_new_state(unsigned char uc_direction)
{
    // Rotation direction is forward
    if(uc_direction == FORWARD)
    {
        // Sector umber is odd
        if(UC_SMC_SECTOR & 0x01)
        {
            // decrement state number
            UC_SMC_STATE--;
            if(UC_SMC_STATE == 255) UC_SMC_STATE = 31;
        }
        // Section umber is even
        else
        {
            // increment state number
            UC_SMC_STATE++;
            if(UC_SMC_STATE == 32) UC_SMC_STATE = 0;
        }
    }
    // Rotation direction is backward
    else
    {
        // Sector umber is odd
        if(UC_SMC_SECTOR & 0x01)
        {
            // increment state number
            UC_SMC_STATE++;
            if(UC_SMC_STATE == 32) UC_SMC_STATE = 0;
        }
        // Sector umber is even
        else
        {
            // decrement state number
            UC_SMC_STATE--;
            if(UC_SMC_STATE == 255) UC_SMC_STATE = 31;
        }
    }
}
```

© 2000 National Semiconductor Corporation

```
//*****************************************************************************
// Function: set_new_sector()
//  Calculate next section of the state-machine
//*****************************************************************************
void set_new_sector(unsigned char uc_direction)
{
   // Rotation direction is forward
   if(uc_direction == FORWARD)
   {
      // increment sector number
      UC_SMC_SECTOR++;
      if(UC_SMC_SECTOR == 4) UC_SMC_SECTOR = 0;
   }
   // Rotation direction is backward
   else
   {
      // decrement sector number
      UC_SMC_SECTOR--;
      if(UC_SMC_SECTOR == 255) UC_SMC_SECTOR = 3;
   }
}
```

5.2 The Module SMC_VTUS.C

This modules contains the dispatch table and a default interrupt handler.

```
/*****************************************************************/
/*  Project:     SMC_VTU                                         */
/*  Module:      SMC_VTUS.C                                      */
/*                                                               */
/*  Author:      Armin Stingl                                    */
/*               1/10/2000                                       */
/*               National Semiconductor, Germany                 */
/*                                                               */
/*   Toolset:    IAR Systems, Embedded Workbench for CR16        */
/*               Version: 1.30A                                  */
/*****************************************************************/
//*********************************************
// Include Files
//*********************************************

//*********************************************
//        Function Prototypes
//*********************************************
extern void cstartup(void);
#pragma fucntion = __interrupt
void reset(void);
extern void T0_IRQ(void);
#pragma fucntion = default

//*********************************************
//     Interrupt Vector Table
//*********************************************
void (*const dispatch_table[]) (void) =
{
    // Traps
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,
```

© 2000 National Semiconductor Corporation

```
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,
    reset,

    // Interrupts
    reset,       // reserved
    reset,       // Flash EEPROM
    reset,       // reserved
    reset,       // reserved
    reset,       // reserved
    reset,       // ADC
    reset,       // MIWU3
    reset,       // MIWU2
    reset,       // MIWU1
    reset,       // MIWU0
    reset,       // USART2 TX
    reset,       // USART1 TX
    reset,       // reserved
    reset,       // MWIRE/SPI
    reset,       // ACCESS.bus
    reset,       // USART2 RX
    reset,       // USART1 RX
    reset,       // reserved
    reset,       // CAN
    reset,       // reserved
    reset,       // reserved
    reset,       // reserved
    reset,       // reserved
    reset,       // VTUD
    reset,       // VTUC
    reset,       // VTUB
    reset,       // VTUA
    reset,       // IRQ27 = T2B
    reset,       // IRQ28 = T2A
    reset,       // IRQ29 = T1B
    reset,       // IRQ30 = T1A
    T0_IRQ       // IRQ31 = Timer 0 Interrupt
};


/*************************************************************/
/*  INTERRUPT: reset                                         */
/*************************************************************/
#pragma fucntion = __interrupt
void reset(void)
{
    __asm__("br cstartup");
}
#pragma fucntion = default
```

5.3 The C-Startup File CSTARTUP.S41

```
   EXTERN    main
   EXTERN    dispatch_table
   RSEG      CSTACK
   RSEG      ISTACK
   PROGRAM   Mystart
   PUBLIC    cstartup

   RSEG      CSTART

cstartup:
```

© 2000 National Semiconductor Corporation

```
    MOVW    $dispatch_table, r0
    LPR     R0, intbase
    MOVW    $SFE(CSTACK), SP
    MOVW    $SFE(ISTACK), R0
    LPR     R0, isp
    BR      main

    END
```

5.4 The Linker Definition File (CR16MCS9.XCL)

```
/*****************************************************************************/
/*                             cr16mcs9.xcl                                  */
/*                                                                           */
/*        XLINK command file for the CR16MCS9                                */
/*        to be used with the CR16 C-compiler (ICCCR16)                      */
/*                                                                           */
/*****************************************************************************/
/*****************************************************************************/
/*      Define CPU                                                           */
/*****************************************************************************/
-ccr16

/*****************************************************************************/
/*      Segments:                                                            */
/*****************************************************************************/

/*************************/
/* C Start-Up Routine   */
/*************************/
-Z(CODE)CSTART=0-13

/*************************/
/* Program Memory       */
/*************************/
-P(CODE)CONST,CODE,RCODE,CDATA0=14-BFFF,1C000-1FFFF

/*************************/
/* User RAM             */
/*************************/
-Z(DATA)IDATA0,UDATA0=C000-CB8B

/*************************/
/* Program Stack        */
/*************************/
-Z(DATA)CSTACK+64

/*************************/
/* Interrupt Stack      */
/*************************/
-Z(DATA)ISTACK+10

/*****************************************************************************/
/*      Select the 'C' library to use                                        */
/*                                                                           */
/*      cl16bs    if option: -ms     (small  mem.model, 4 byte doubles)*/
/*****************************************************************************/
-C cl16bs

/*****************************************************************************/
/*      End of File                                                          */
/*****************************************************************************/
```

© 2000 National Semiconductor Corporation

6.0 CPU Utilization and Code Size

By using the optimization option of the IAR Systems toolset (optimize for size) the implementation presented in chapter 4 requires less than 700 bytes of program memory, including the two look-up tables. Furthermore, only 6 bytes of RAM are needed to hold the values ofl variables.

At a main system clock of 16MHz and at a rotation speed of 2000 micro-steps per second, the CPU utilization of the stepper motor controller state-machine (T0 interrupt) is about 1.5%.

7.0 Increasing the Current Drive Capabilities of the Stepper Motor Controller

The I/O ports of the CR16MCS9 can only drive an output current of +/- 1.6mA. However, most stepper motors have a much higher current consumption, typically above 20mA. In order to meet these requirements external drivers may be connected between the device I/O pins and the stepper motor.
The example, shown in figure 8, uses an octal bus transceiver 74AC245 to drive two stepper motors with a maximum current consumption of 20mA per coil.

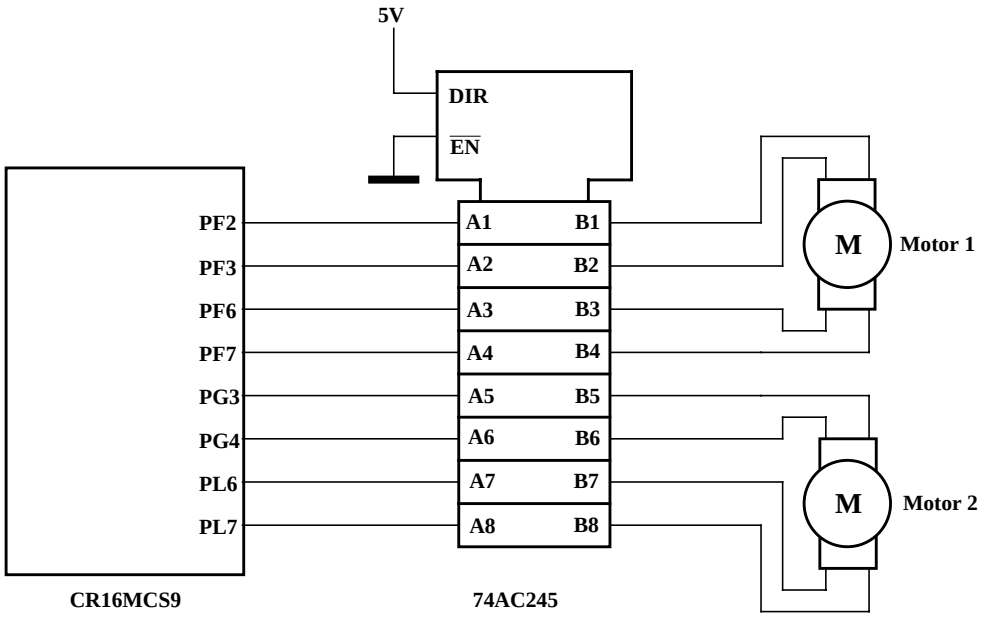

Figure 8. External Driver Connection

© 2000 National Semiconductor Corporation