MOTOROLA
intelligence everywhere™

digital dna™

# Vacuum Cleaner Reference Design

## Designer Reference Manual

M68HC08 Microcontrollers

DRM010/D
Rev. 0, 2/2003

MOTOROLA.COM/SEMICONDUCTORS

# Vacuum Cleaner Reference Design

## Designer Reference Manual — Rev 0

by: Ken Berrenger
Motorola, East Kilbride

[redacted]

# List of Paragraphs

# Table of Contents

## List of Paragraphs

## Table of Contents

## List of Figures

## List of Tables

## Section 1. Vacuum Cleaner Reference Design

## Section 2. Reference Design Code



## Section 3. Revision History



## Section 4. Glossary

# List of Figures

# List of Figures

# List of Tables

| Table | Title | Page |
|---|---|---|

# List of Tables

# Section 1. Vacuum Cleaner Reference Design

## 1.1 Contents



## 1.2 Introduction

The modern vacuum cleaner is an indispensable household appliance. The upright and the canister are the two basic types. The upright is the most common in the United States and the United Kingdom, while the canister vacuum is more common on the European continent. Other kinds of vacuum cleaners — such as small, hand-held vacuums, large, central vacuums, and industrial, floor care appliances — are not addressed in this document.

A vacuum cleaner uses a universal motor meaning that the motor can operate from either an ac or dc supply. It has brushes like a dc permanent magnet motor. However, it has a wound stator that is connected in series with the rotor windings. In dc applications, it is often called a series wound dc motor. A universal motor is used in vacuum cleaners because it can operate at very high speeds. Vacuum cleaner motors operate at speeds up to 30,000 RPMs. The high-speed operation is necessary to generate a strong suction using a small fan. An induction motor is limited to speeds below 3600 RPMs.

Many vacuum cleaners have just a simple on-off switch for the motor control. The penetration of electronic controls in vacuum cleaners is higher in Europe and Asia. Most European canister vacuums have a variable suction power knob or slider. The motor speed is controlled using a triac with a simple firing control circuit. The firing control circuit consists of a few discrete components, such as a diac, resistor, capacitor, and potentiometer.

Today, a few of the more expensive high-end vacuum cleaner models use microcontrollers (MCU). Microcontrollers are used to provide added features for these sophisticated models, such as infrared or wired remote control, status LEDs (light-emitting diode), and automatic suction control.

In the near future, all vacuum cleaners might include a microcontroller, since MCUs provide several benefits for the low-end models.

**Figure 1-1. Vacuum Cleaner Plus Circuit Board**

One of the most important benefits is soft-start. A microcontroller can provide a soft-start function using a very simple software algorithm that will minimize the startup current of the vacuum cleaner. The startup current of a conventional vacuum might be as high as 60 amperes peak. If the vacuum draws excessive current during startup, this might cause the line voltage to dip momentarily. This is readily apparent in common incandescent lighting and is called voltage flicker. With increased European community regulations, vacuum cleaner manufacturers must clean up their power quality. Using an MCU with some simple software will help manufacturers meet the requirements of EN61000-2-3 and EN61000-3-3. These standards define limits for line harmonics and voltage flicker.

## 1.3 System Design

A simple vacuum cleaner reference design is shown in Figure 1-2, and a brief description of the circuit operation follows.

The HC908KX8 is used to generate the triac drive waveform and control the speed of the motor. A potentiometer is used to vary the speed of the motor. The MCU reads the potentiometer using one of the analog-to-digital converter (ADC) port pins. A single port pin is used with a timer input capture function to measure the ac line frequency and sync to the ac line. The current injection into the MCU is limited, using a large value resistor. Four port pins provide sufficient sink current to drive the triac directly. A charge pump power supply is used to provide power for the MCU and drive current for the triac. This type of power supply is useful only up to about 20 mA. The supply current is limited by the size of the ac line capacitor. A high-voltage non-polar capacitor is needed to generate the ac current. A low-cost charge pump power supply does not have sufficient current capability to drive status LEDs. A sensitive gate triac can be used to minimize triac drive current. However, sensitive gate triacs have a lower rate of voltage rise equal dv/dt rating and require a more expensive snubber circuit.

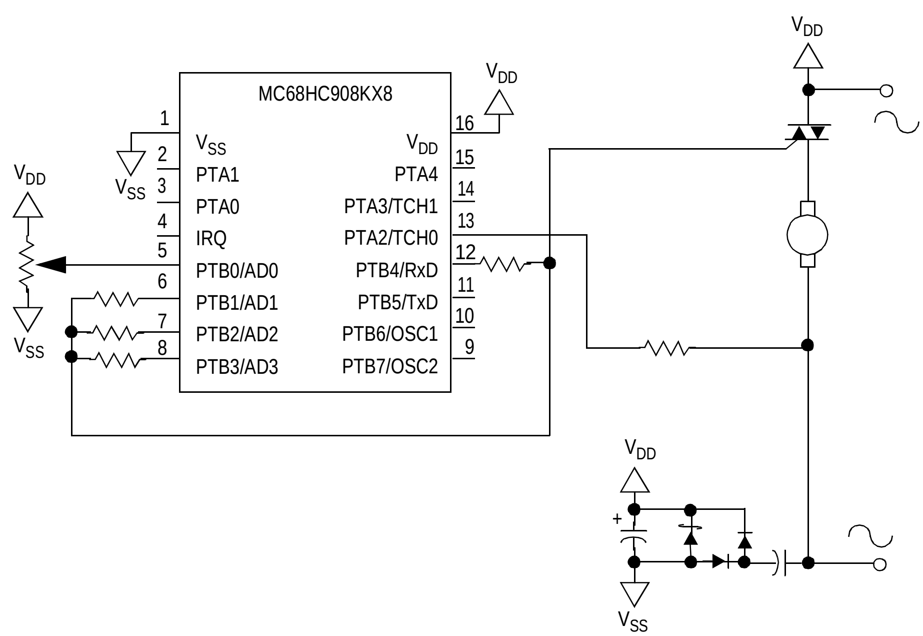


**Figure 1-2. Vacuum Cleaner Reference Design**

While this circuit is simple and cost effective, it is difficult to develop and debug software in this configuration. A circuit has been developed for the express purpose of developing vacuum cleaner software for the HC908KX8. This circuit is shown in Figure 1-6.

A separate isolated supply provides power to the MCU and triac drive. This provides safe isolation when working with a Motorola modular development system (MMDS). This allows a safe direct connection from MMDS to the MCU socket using a flexible cable. The software may be safely debugged without connecting the triac to the ac mains. An external oscillator is used when programming or communicating via monitor mode.

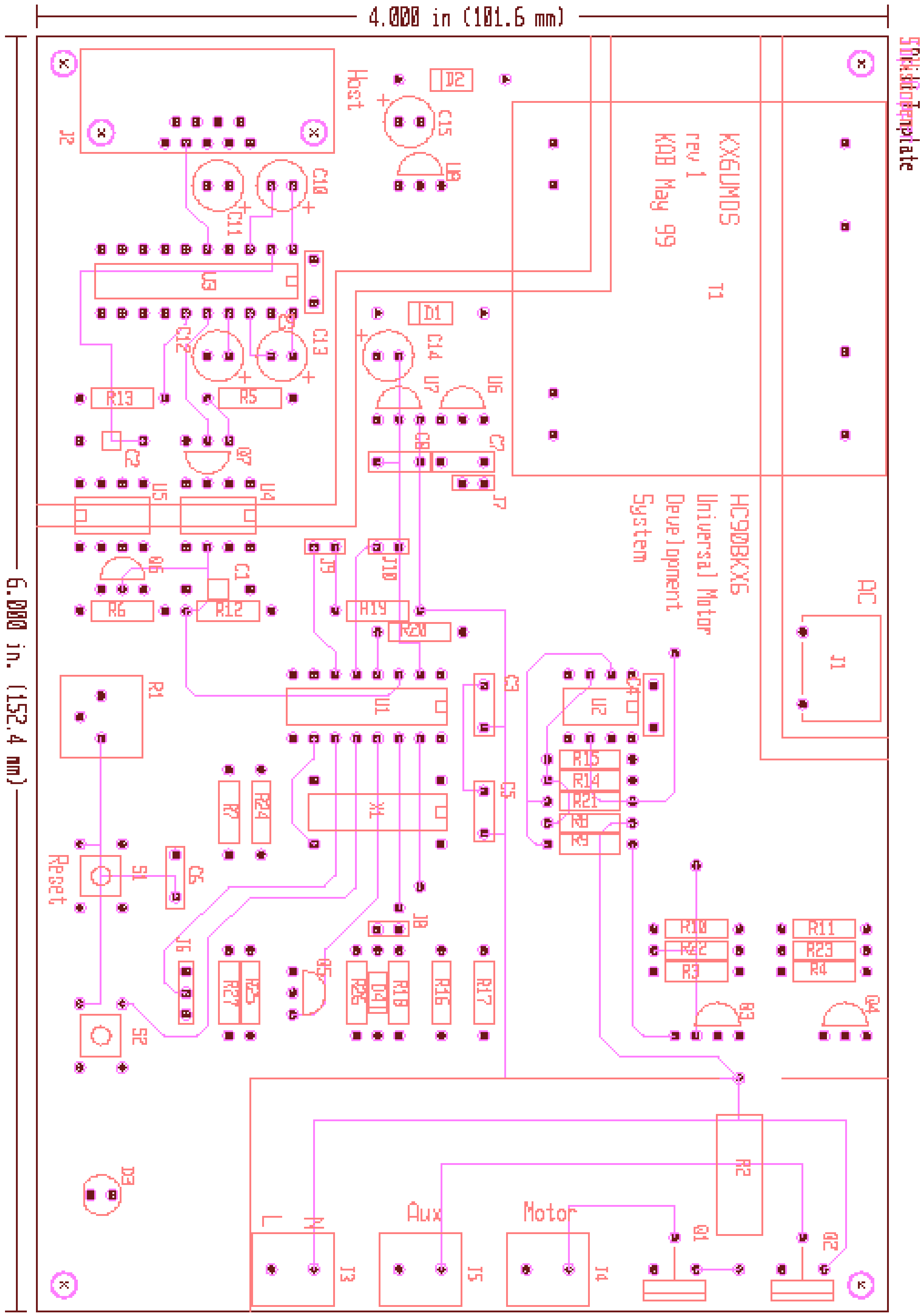


Figure 1-3. HC908KX6 Universal Motor Development System (top)

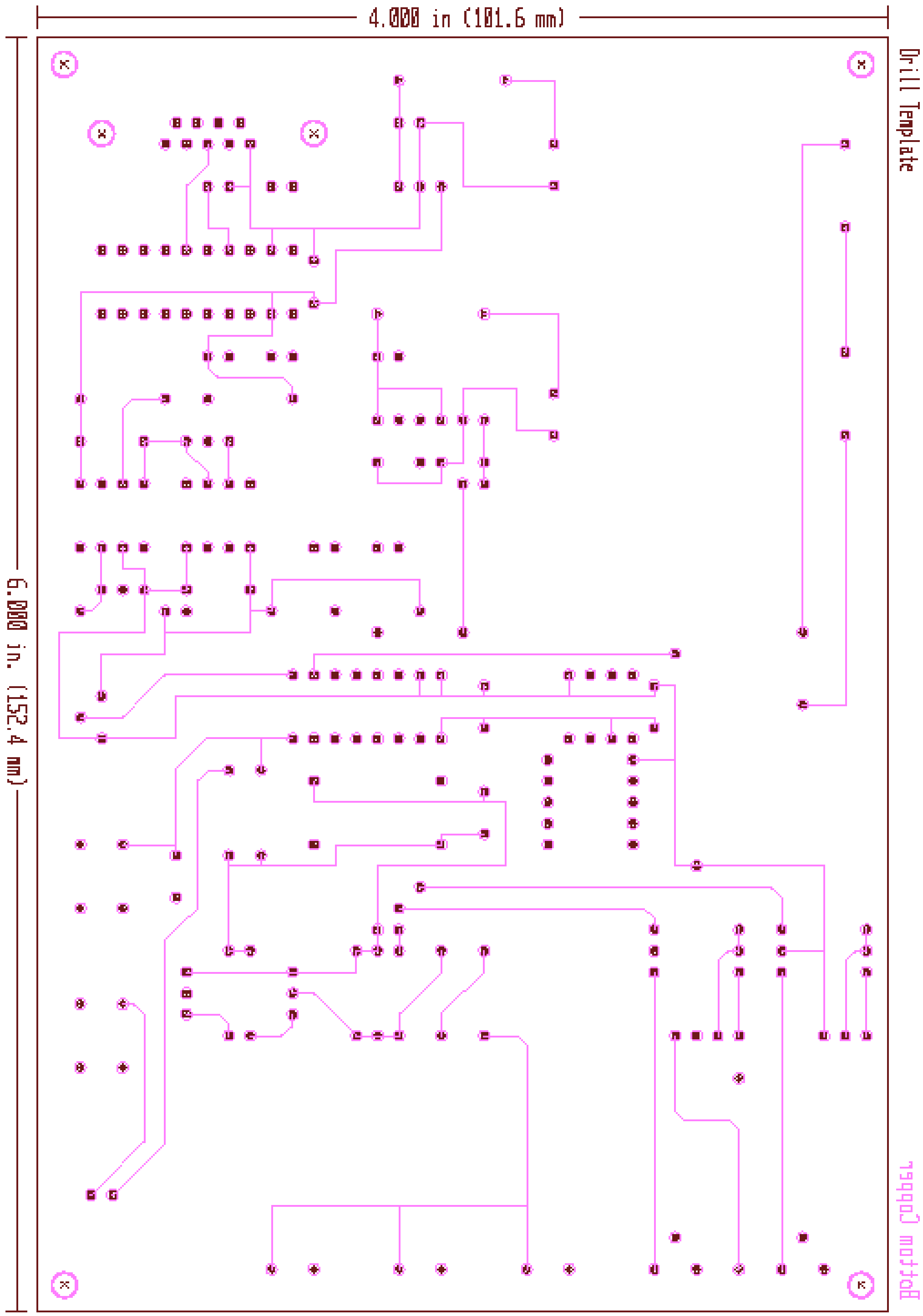


**Figure 1-4. HC908KX6 Universal Motor Development System (bottom)**

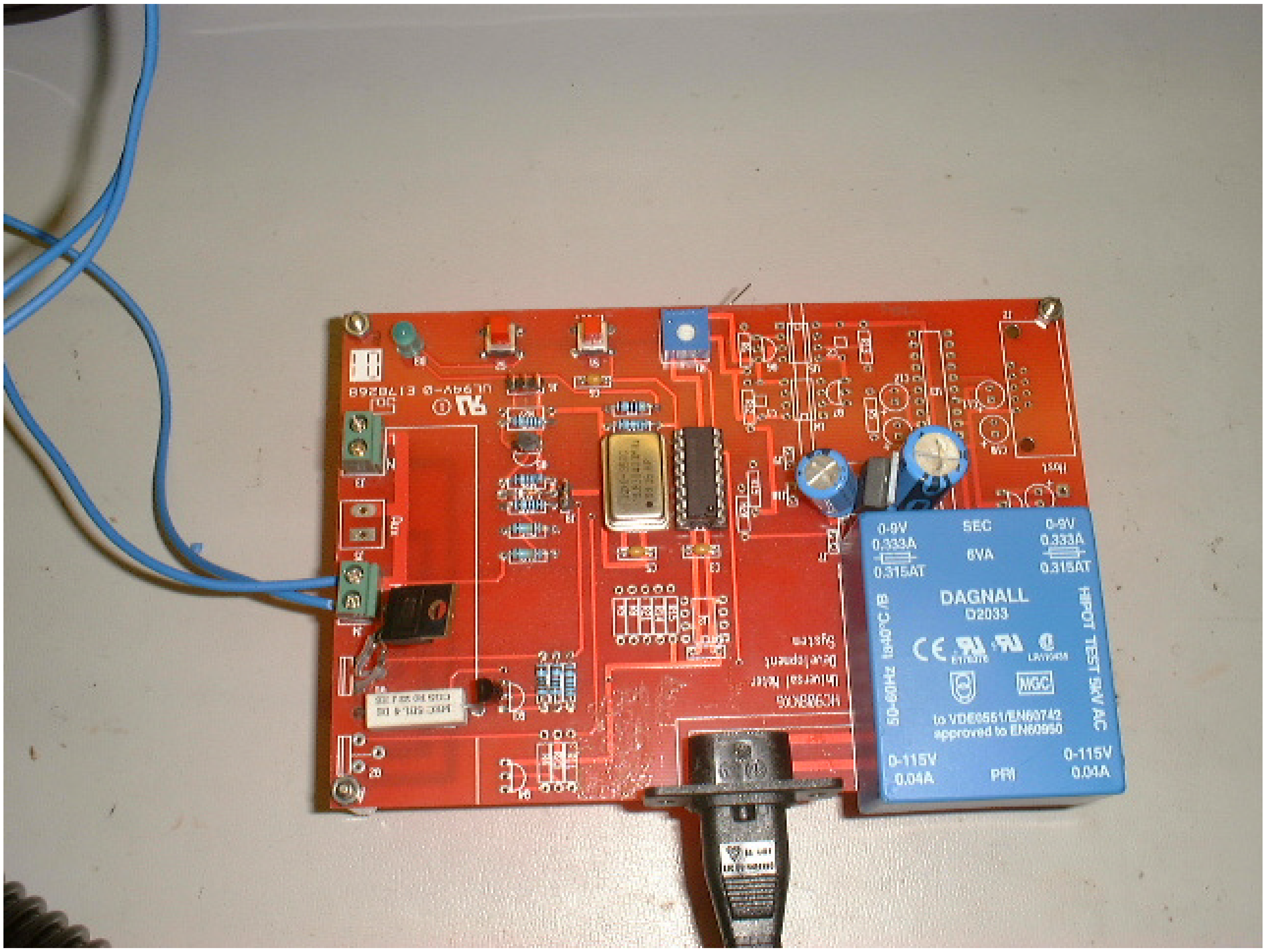


**Figure 1-5. Vacuum Cleaner Circuit Board**

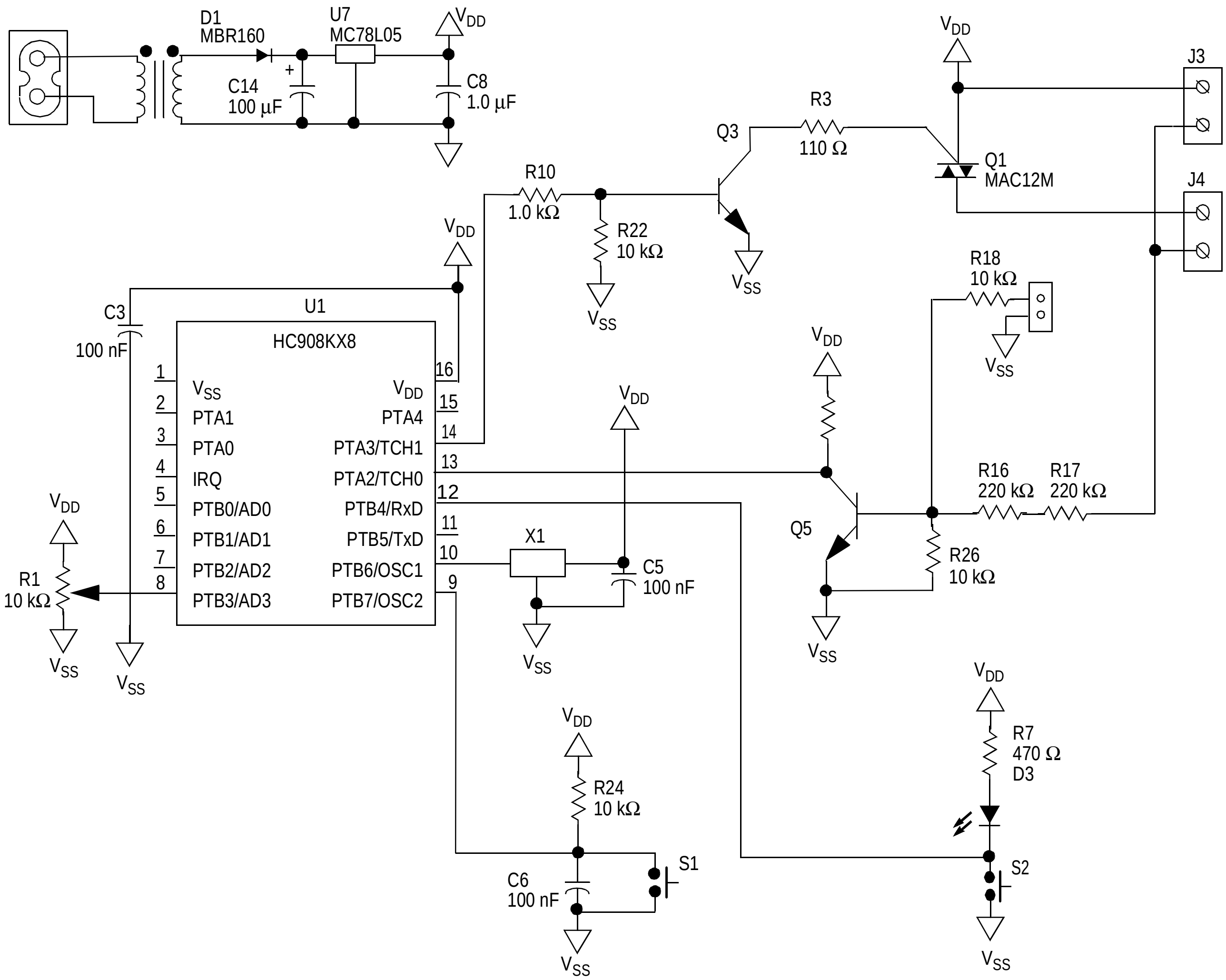


**Figure 1-6. Vacuum Cleaner Software Development System**

While it is possible to drive the triac directly, an NPN drive circuit has several advantages. The NPN drive circuit consists of Q3, R3, R10, and R22 in Figure 1-6. The NPN drive circuit uses only one port pin. Using the output compare pin of the timer provides accurate hardware-generated timing. Multiple pins can be manipulated only by using software and they have a resulting interrupt latency. The NPN transistor drive also has better EMC (electromagnetic compliance) robustness than a direct drive solution. The MCU is not exposed to current injection due to the triac gate drive voltage. The cost of a small NPN is less than the cost of a good Schottky diode or zener diode that are commonly used for EMC protection.

An additional NPN transistor (Q5) is used to provide an accurate zero voltage crossing detection circuit. The zero voltage crossing circuit outputs a square wave to the MCU input capture. The NPN provides a square wave output over a wide range of input voltages. Two or more resistors in series may be required due to the limited voltage rating of metal film or chip resistors. A jumper is also included, providing a convenient connection to a pulse generator for debugging purposes.

One of the port pins is used to drive an LED. This is used to indicate the software has synchronized with the ac line to aid with debugging.



## 1.4 Phase Angle Control Basics

The concept of phase angle control is to apply only a portion of the ac waveform to the load. This is illustrated in Figure 1-7. Once fired, the triac will conduct until the next zero crossing. The average voltage is proportional to the shaded area under the curve. The phase angle is measured from the trigger point to the next zero crossing. This is also referred to as the conduction angle or firing angle.

The phase angle is varied continuously and results in a variety of voltage waveforms. This is illustrated in Figure 1-8. The phase angle control software should be able to smoothly vary the phase angle to control the average voltage applied to the load. Rotating the potentiometer should increase the phase angle and use a larger portion of the sine wave.

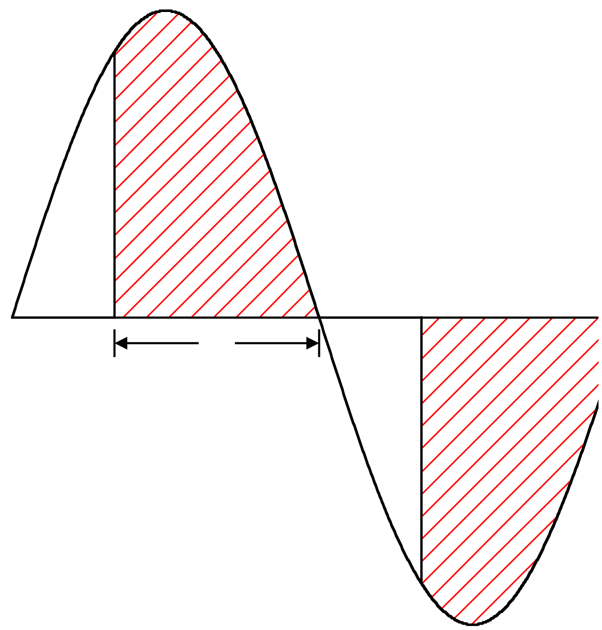

**Figure 1-7. Phase Angle Control**

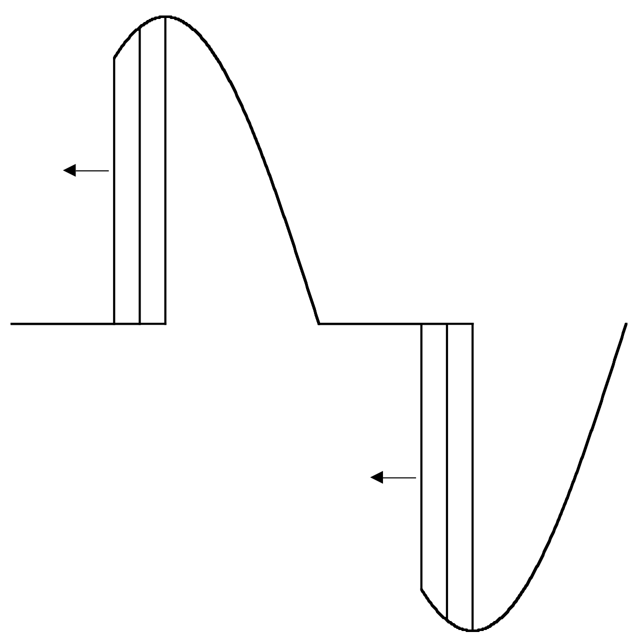

**Figure 1-8. Continuously Variable Phase Angle**

This simple control method is sufficient for most universal motors and other loads. More complex forms of phase angle control compensate for the inductive load or provide sensorless speed control of a universal motor. Closed loop speed control might use a PID (proportional integral derivative) loop or fuzzy logic algorithm. A vacuum cleaner normally does not require this level of complexity.

## 1.5 Triac Drive Waveform

The key task of the phase angle control software is to provide the trigger pulse for the triac. The software must synchronize to the ac line voltage and fire the triac at the desired angle. The design of the triac firing pulse requires some basic knowledge of the operation of triacs.

Triacs are a latching bilateral switching device. When the triac is off, it will block voltage in both directions. Once a triac has been fired, it will latch in the on state and continue to conduct until the current decreases to zero. The current for an ac load naturally crosses zero every half cycle. Zero current turn off is in fact desirable and minimizes any inductive kickback voltage. The triac will not latch on until after the voltage has increased to above its rated latching voltage and the current has increased to greater than its rated latching current. Once latched, the triac will stay on until the current has decreased to below the rated holding current. For these triac specifications, contact ON Semiconductor at http://www.onsemi.com. The document order number is MAC12SM/D.

Because the current passes through zero, the triac must be refired every half cycle. The triac is fired by applying a trigger pulse to the gate terminal. A negative gate current is desired for most triacs because the trigger current is much higher using a positive trigger, in particular when the load voltage is negative. The duration of the trigger pulse must be long enough for the load current to reach the triac's rated latching current. Once the triac has latched, there is no need to continue to supply trigger current.

The ac line voltage zero crossing is easily measured using a simple circuit as shown in Figure 1-6. The load current is not so easily measured. The MCU's ADC requires a 0- to 5-V analog signal. Accurately measuring the load current would require a very small value resistor and an operational amplifier with a low input-offset voltage. Fortunately, the load current is not really needed for most applications. Assume that the load current will lag the voltage for most inductive loads. Vacuum cleaner universal motors are highly inductive.

When driving an inductive load, the current will lag the voltage. The triac does not turn off at zero-voltage crossing. It will continue to conduct for some degrees until the current passes through zero. As the phase angle is increased, at some point, the current will become continuous. The triac will be fired just after zero current crossing. As the phase angle is increased further, approaching 180°, the triac will be fired before current zero crossing. If a short pulse is used at these angles, the triac will not conduct over the rest of the cycle, as shown in Figure 1-9. The end result would be that the motor suddenly slows when the speed is turned all the way up.

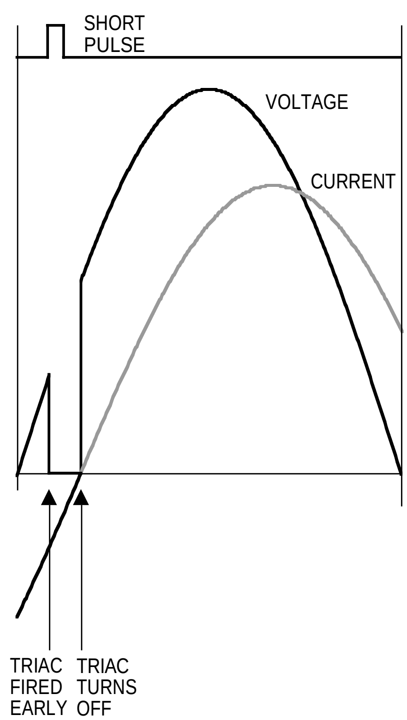


**Figure 1-9. Undesirable Turn Off Using Short Pulse**

This potential problem can be remedied by extending the triac pulse as the phase angle approaches 180°. Extending the triac pulse out to about 135° will accommodate inductive loads with a current phase angle up to 45°. This is suitable for most applications.

Extending the triac pulse is illustrated in Figure 1-10.

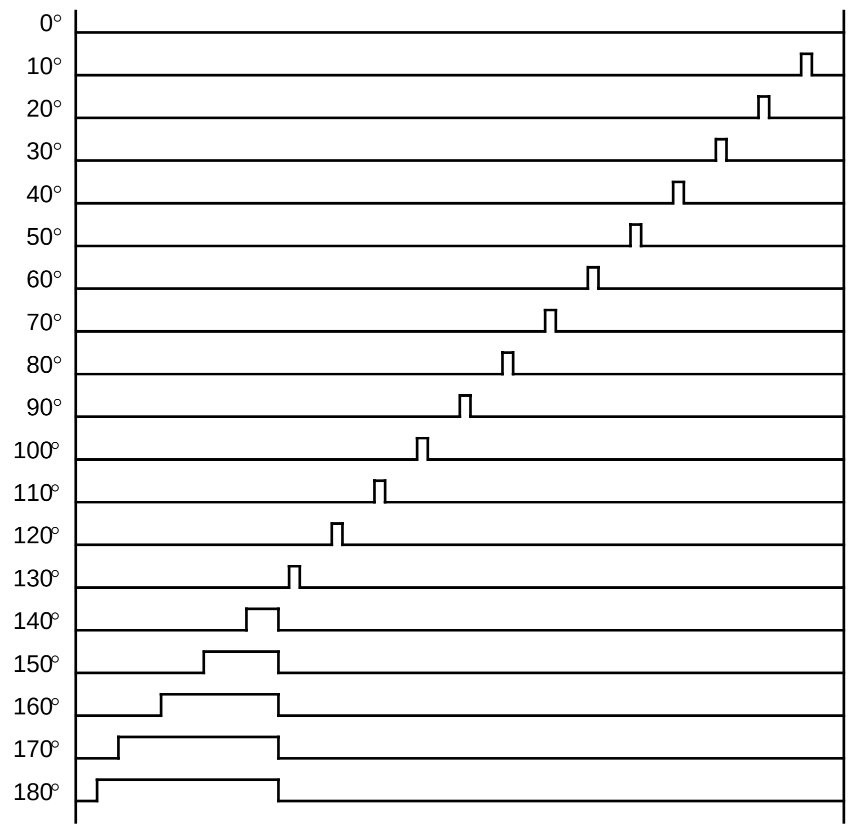


**Figure 1-10. Triac Pulse at Different Phase Angles**

Zero degrees and 180° also require special attention. At zero speed, the desired output is zero voltage and the triac should not be fired. Firing the triac at zero degrees would give full speed operation. If the desired phase angle is too small, the triac should also not be fired. If the triac pulse were to overlap the next voltage zero crossing, the voltage would jump from 0 to 100 percent, resulting in an undesirable plugging of the motor.

However, small phase angles are needed in some applications. A vacuum cleaner universal motor has a very low impedance when the rotor is not moving. If the initial voltage is too high, a high current surge will result. Experiments have shown that a minimum phase angle of about 5° is low enough to provide smooth starting. This is essential to minimize the startup surge current.

Because the current is lagging the voltage, full speed will occur before 180°. The last few degrees do not provide any variation in the motor speed. It is not crucial to go all the way to the zero crossing. Some delay between the zero crossing and triac firing is generally acceptable for inductive loads.

The desired triac firing pattern is summarized in the Table 1-1.

**Table 1-1. Triac Pulse Generation**

| Phase Angle | Action |
|---|---|
| 0° < ϕ < 5° | None |
| 5° < ϕ < 135°° | Short pulse at ϕ |
| 135° ϕ < 175° | Turn on at ϕ; turn off at 135° |
| ϕ > 135° | Turn on ASAP; turn off at 135° |

## 1.6 Vacuum Software

Software has been developed for basic vacuum cleaner universal motor control. This software was developed for the HC908KX8. The software will run on any HC08 MCU that has at least a 2-channel timer and an ADC. The HC908KX8 also features an internal oscillator and a small 16-pin package. The software is compatible with the internal oscillator or other low-cost RC oscillators.

The 2-channel timer provides all the necessary timing control for the software. One channel is used for an input capture to measure the ac line zero crossing. The second channel is configured as an output compare and is used to control the timing of the triac pulse.

The software is written in C language. There are numerous advantages to programming in C, even for small microcontrollers. For instance, the

HC08 has stack manipulation instructions that permit the C compiler to effectively use local variables and minimize the RAM requirements. The HC08 also provides very good code efficiency due to the short instruction length. The HC908KX8 has eight Kbytes of FLASH memory, more than enough for a small program. The phase angle control software takes only about 1.2 Kbytes of program memory. Even a small 1.5-k part might be programmed in C if no complex math or library functions are included. For this project, the HIWARE C compiler was used to produce compact code comparable in code size to a hand-coded assembler in many instances.

The software can be organized in several different ways. For example, the code could be written as a straightforward procedure using polling. When using polling, the software would test or poll the zero crossing pin and wait for a zero crossing. This is the preferred method when using a small HC05 MCU with limited peripherals. Using an HC08 with a 2-channel timer, the software can be written using interrupts. This provides more time for the CPU to perform other functions.

Once the MCU has been initialized, all processing could be done in interrupt service routines. This is a common method of organizing software. The main procedure would end with a while(1) statement and all processing is handled by the ISRs (interrupt service routine).

The zero crossing and triac pulses are time critical events and are best handled by the hardware timers and serviced using interrupts. Other functions are not time critical and could be performed anywhere in the ac cycle.

The control loop functions such as reading the ADC, scaling, integrating, and saturation are not time critical. These functions can be placed in the main loop. The interrupt service routines will interrupt the calculations as needed. A mechanism is then needed to synchronize these functions to the ac line. A sync flag is used for this purpose. The main loop will wait for the sync flag before updating the phase. The input capture routine will set the sync flag, enabling the main loop functions. The main loop will then update the phase information and clear the sync flag.

The resulting flowchart is shown in Figure 1-11. This is a combined flowchart and state diagram.

Once the MCU is initialized, the main loop may be interrupted by the input capture and the output compare interrupt service routines.

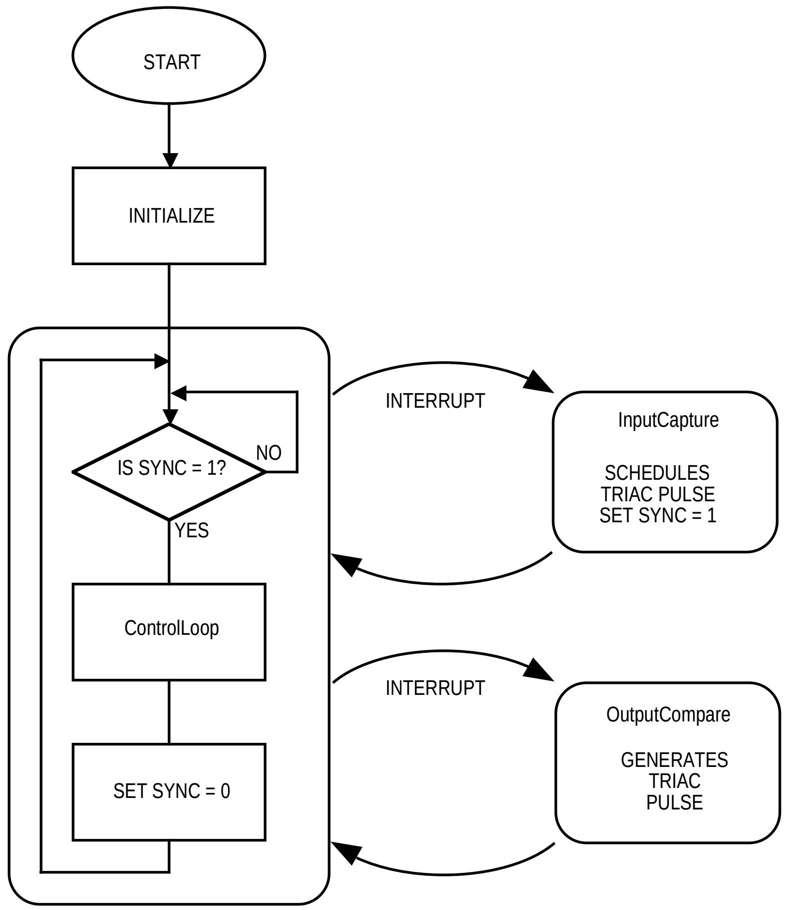


**Figure 1-11. Software Flow Diagram**

## 1.7 Input Capture

The input capture interrupt service routine is executed every time the input capture timer channel detects an edge on the zero crossing input. The input capture time is read from the timer channel. The average period is calculated over two cycles. This value is used as criteria for locking on the input line frequency. The minimum and maximum periods are set up to lock from 30 Hz to 90 Hz. This will accommodate both 50- and 60-Hz possibilities and up to 50 percent error in the clock frequency.

An LED is used for debugging to indicate when the software has locked onto the line frequency.

The triac pulses are scheduled by the input capture interrupt service routine.

- If the phase is less than 5°, the triac is not pulsed.
- If the phase is less than 135°, a short pulse is used.
- If the phase is greater than 135°, a long pulse is used.

The pulse function calculates the desired rising and falling edges for each case. The long pulse function also checks the current time. If the desired time has already expired, the rising edge will be scheduled as soon as possible (ASAP). The input capture function sets up the output compare to generate the rising edge. The time for the falling edge is calculated and saved for use by the output compare function.

## 1.8 Output Compare

The output compare interrupt service routine is called for both rising and falling edges of the triac pulse. The output compare fires the triac and also turns off the gate pulse at the pre-determined time. If the output compare was called as a result of the output compare being set high, the low edge will be scheduled using the saved off time. Otherwise, the output compare will be disabled.

This method of using the output compare to schedule subsequent output compares is extremely powerful. Using software, the output compare can be used to generate practically any desired series of pulses. Once initiated, the output compare can generate these pulses autonomously.

## 1.9 Control Loop

The control loop function provides the basic motor control features.

First, the potentiometer is read using the ADC. This takes some time and program execution will wait until the conversion complete flag has been set. The analog-to-digital reading is from 0 to 255. The ADC also has some uncertainty. The bottom and top range of the potentiometer setting should provide zero and full-speed operation. Saturation is provided to compensate for this requirement.

The ADC value is scaled to obtain the desired 0° to 180° range. The ADC reading is multiplied by 180, then divided by 255. The HC08 is capable of performing an 8 by 8 multiply and a 16 by 8 divide efficiently. Inline assembler functions are used to access the HC08 math functions directly, resulting in fast and efficient code. Most ANSI C compilers (American Standard Code for Information Exchange) will promote both operands to 16-bit integers before performing a multiply or divide. This results in inefficient code when using 8-bit unsigned char variables.

An integral controller is used to provide soft-starting and a smooth ramp in the motor speed. Slowly, the controller will increment the output phase until it reaches the desired speed setting. This will limit the motor current during starting. When the desired speed is modified by changing the potentiometer setting, the speed will smoothly ramp to the new setting.

The integrate function provides a simple integral controller. The integrator output is stored in a static variable called PhaseI. This variable is incremented or decremented depending on the input variable. The integrator update rate determines how fast the integrator will ramp the output.

The saturate function compensates for the characteristics of the potentiometer and the desired output phase. If the input is less than 5°, the output will be rounded down to 0°. If the input is greater than 175°, it will be rounded up to 180°. The saturation is placed after the integrator to provide saturation to the output phase.

## 1.10 Interrupt Timing

The input capture always occurs at the zero crossing. Normally, the MCU should be idle at this time. The timing of the output compare interrupt service routine is variable and depends on the phase angle and the width of the triac pulse. If the phase angle is small, a short pulse is used, and the output compare function is called twice in rapid succession. The triac pulse must be large enough to account for the interrupt latency of the output compare function.

If the phase angle is larger than 135°, the output compare will be called first at the phase angle and then a second time at 135°. There is an idle period between the output compare interrupts.

The control loop function is normally executed after the input capture function. However, when the triac phase angle aproaches its maximum of 180°, the output compare interrupt service routine will pre-empt the control loop. The time of the interrupt service routines is shown in Figure 1-12.

The MCU is idle for most of the ac cycle. The time critical operations occur immediately after zero crossing. The execution time of the input capture routine will determine the maximum phase angle. A maximum phase angle of 175° is perfectly acceptable for an inductive load like the vacuum cleaner motor.

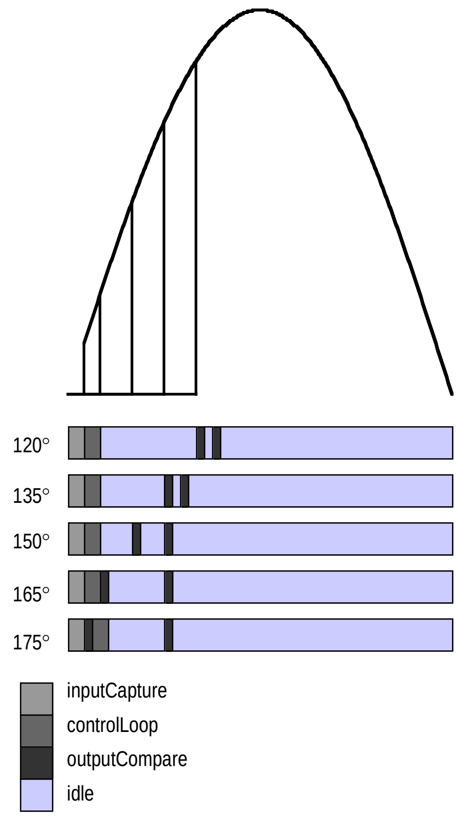


**Figure 1-12. CPU Process Timing**

## 1.11 Results

The software was first tested using a modular microcontroller development system (MMDS), bus state analyzer, pulse generator, and digital oscilloscope. The pulse generator was used to simulate the ac line and test the lock range of the software. The digital storage scope was used to examine the triac pulse waveform in relation to the pulse generator. The first few pulses are critical to the startup operation. It is important that there be no errant pulses. The pulses also should change smoothly without any glitches.

Once fully satisfied with the operation of the software, a FLASH device was programmed and inserted in the socket. The test board was connected to a vacuum cleaner motor and an ac variac and safety isolation transformer. Bringing the ac line up slowly will minimize the chance of catastrophic failures due to wiring or software errors. Later, the system was tested for hard-starts using an on-off switch directly off a stiff ac source.

After debugging, the software provided good results. A startup delay was added to prevent errant pulses during startup. The integrator rate was adjusted to provide a ramp time of about three seconds from zero to full speed. This provided a smooth soft-start and a good feel to the speed control.

The startup current of the vacuum cleaner motor was measured using a simple on-off switch shown in Figure 1-13. The peak current was about 40 amps. The startup current was measured using the same motor with the electronic soft-start shown in Figure 1-14. The peak current was less than 20 amps. The performance is better than the numbers indicate. In fact, the MCU controlled motor did not exhibit a startup surge at all. The peak current occurs when the motor reaches maximum speed.

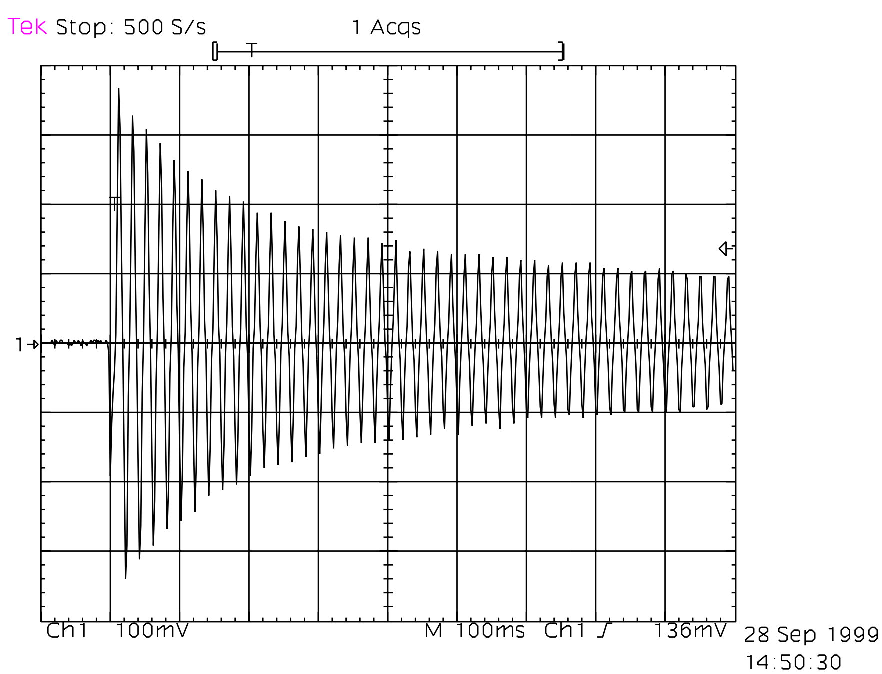


**Figure 1-13. Hard Start Using an On-Off Switch**

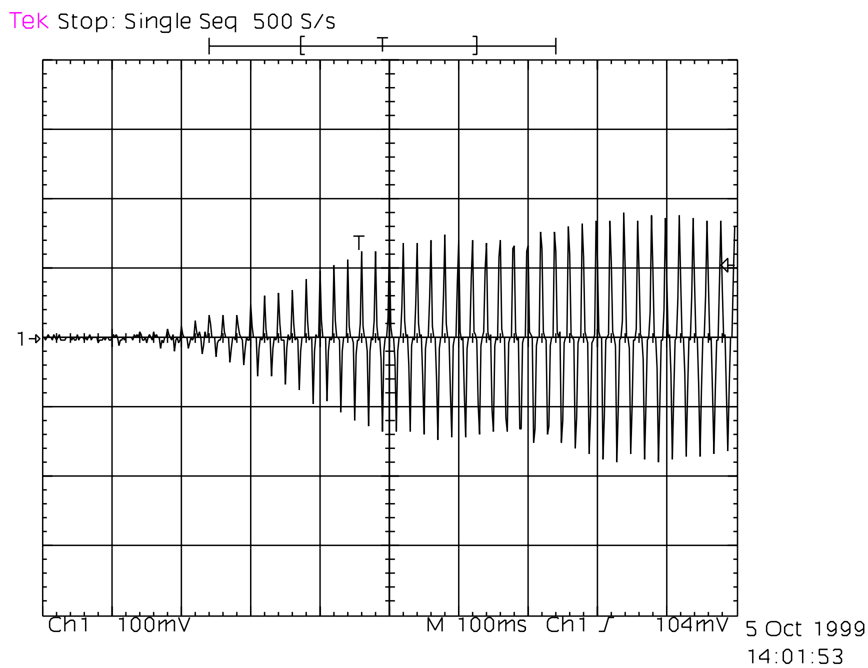


**Figure 1-14. Soft Start Using MCU with Software**

The soft-start feature effectively eliminates any startup surge. The software provides robust control of a universal motor over a wide range of conditions. The software is compact, efficient, and suitable for any HC08 microcontroller.

## 1.12 Conclusions

A cost-effective microprocessor-based system has been developed to provide phase angle control for vacuum cleaners. Software has been developed to provide phase angle control with a soft-start feature. This solution has proven to dramatically reduce the startup current. The reduction in startup current is essential to meet increasingly stringent power quality requirements in Europe.

The basic requirements on phase angle control and triac drive have been discussed. The software provides optimum pulse generation for driving inductive loads. The pulse width is lengthened for large phase angles.

The 8-bit microcontroller chosen, the Motorola HC908KX8, provides all of the required features in a small 16-pin package. This internal oscillator eliminates the need for an external crystal or RC oscillator. This device is well suited for vacuum cleaners and other small appliances.

The basic software coded in C, when compiled, only uses about 1200 bytes of program memory. This leaves about 7000 bytes of program memory available for additional functions. The C code listing is included in this application note and follows the main text. The interrupt-driven software uses less than 10 percent of the total available CPU (central processor unit) load running at 4 MHz. This frees the CPU for other tasks. Additional functions can be implemented in the foreground with good performance virtually unimpeded by interrupt processing. The software is flexible and reusable for a variety of phase angle control applications.

# Section 2. Reference Design Code

## 2.1 Contents



## 2.2 Vacuum Cleaner Reference Design Code

```
/*******************************************************************************
                                                   Copyright (c) Motorola 1999
File Name         :  vacuum.c

Engineer          :  Ken Berringer

Location          :  EKB

Date Created      : 1 Dec 1999

Current Revision :  1.0

Notes:

    This is the code for the vacuum cleaner reference design. The code
    includes detailed comments before each function. The code is
    organized in a single C file. There are two included header files,
    one for the standard HC08KX8 register definitions "hc08kx6.h" and
    a second application specific header file for the vacuum code
    "vacuum.h". All of the function prototypes and constants are in
    the vacuum.h header file.

    Most of the code is in high level C code. Hardware driver
    functions are in low level C or inline assembler.

*******************************************************************************/

#include "hc08kx6.h"
#include "vacuum.h"


/******************************************************************************
```

```
global variables

    The following variables are defined as global. These are used and
    modified by both the main loop and the interrupt service routines.
    There are manually initialized and don't depend on the startup
    code.

*****************************************************************************/

unsigned char Phase;
unsigned int Degrees;
unsigned char Sync;
unsigned int OffTime;

/*****************************************************************************

Global Variables (could be static)

    The following variables are defined as global. They are only used
    in specific functions and could also be declared as static. Global
    variables are more useful for debugging. The contents of global
    variables are listed in the debugger, static variables are not
    listed until the function is entered. These variables are also
    initialized manually by the init function.

*****************************************************************************/

unsigned char Update;
unsigned char PhaseI;
unsigned int Period[2];
unsigned char Cycle;
unsigned int Told;
signed char ModError;

/******************************************************************************

function    :   main()

parameters  :   void

returns     :   void

type        :   main


Description:
    The main code is very simple. It initializes everything, then
    calls a startup delay. The endless while(1) loop will wait until
    after the input capture sets the sync flag. Then it update the
    phase angle and reset the flag.
```

```
******************************************************************************/
void main (void)
{
    init();
    startupDelay(8);
    while(1)
    {
        while(!Sync);
        Phase = controlLoop();
        Sync=0;
    }
}

/******************************************************************************

function    :   init()

parameters  :   void

returns     :   void

type        :   normal


Description:
    The init function clears all global variables explicitly. This
    allows one to eliminate the startup code. It then initializes the
    timer, enables the input capture, and enables interrupts.

******************************************************************************/

void init (void)
{
    initMCU();
    unsigned char Phase=0;
    Phase=0;
    Degrees=0;
    Told=0;
    Period[0]=0;
    Period[1]=0;
    Cycle=0;
    Update=0;
    ModError = 0;
    Sync=0;
    OffTime=0;
    initTimer();
    enableIC();
    ENABLE_INTERUPTS;
}

/******************************************************************************
```

```
function     :   startupDelay()

parameters   :   unsigned char i - delay count, zero crossings

returns      :   void

type         :   normal re-entrant


Description:
    The startupDelay() function is provided to ensure that the
    everything is stable before starting the motor. This prevents
    errant pulses which might result in a high surge currents. The
    Phase angle will remain at zero until the startupDelay is
    completed.

********************************************************************************/
void startupDelay(unsigned char i)
{
    unsigned char j;

    for (j=0;j<i;j++)
    {
        while(!Sync);
        Sync=0;
    }
}

/*******************************************************************************

function     :   controlLoop()

parameters   :   void

returns      :   void

type         :   normal re-entrant


Description:
    The controlLoop function is executed after each zero crossing.
    This function reads the pot updates the phase angle. Scaling
    integration and saturation are implemented in this function.
    A more complex PID loop or fuzzy logic block could also be
    implemented here. The function is executed with interrupts
    enabled and might be interrupted by the output compare function.


********************************************************************************/
```

```
unsigned char controlLoop(void)
{
    unsigned char x;
    x = readPot();
    x = scale (x);
    x = integrate(x);
    x = saturate(x);
    return x;
}

/****************************************************************************

function    :   scale()

parameters  :   unsigned char x - input from the A/D

returns     :   unsigned char x - scaled output

type        :   normal re-entrant


Description:

    The scale function will scale the A/D measurement (0-255) to
    degrees (0-180). The number is first multiplied by 180, then
    divided by 255. The mul() and div() functions are optimized for
    HC08 8-bit math.

****************************************************************************/
unsigned char scale(unsigned char x)
{
    unsigned int y;

    y = mul(180,x);
    x = div(y,255);
    return x;
}

/****************************************************************************

function    :   integrate()

parameters  :   unsigned char x - input

returns     :   unsigned char x - output

type        :   normal re-entrant


Description:
```

```
    The integrate function provides a simple integral controller to
    provide a smooth ramp to the phase output. The UPDATE_RATE sets
    the rate at which the integrator output is incremented. Update and
    PhaseI could be global or static.

*******************************************************************************/

unsigned char integrate(unsigned char x)
{

    Update++;

    if(Update==UPDATE_RATE)
    {
        Update = 0;
        if (x > PhaseI)
        {
            PhaseI++;
        }
        else if (x < PhaseI)
        {
           PhaseI--;
        }
    }

    return PhaseI;
}

/*******************************************************************************

function     :   saturate()

parameters   :   unsigned char x - input

returns      :   unsigned char x - output

type         :   normal re-entrant


Description:

    This function provides saturation for the output phase. If the
    phase is greater than 175 or less than 5, it is rounded up or down
    respectively.

*******************************************************************************/

unsigned char saturate(unsigned char x)
{
    if (x < 5)
    {
```

```
        x = 0;
    }
    else if (x > 175)
    {
        x = 180;
    }
    return x;
}

/*******************************************************************************

function     :   inputCapture()

parameters   :   void

returns      :   void

type         :   interrupt service routine


Description:

    This function is called every time a zero crossing occurs. It
    reads the input capture register and calculates the average
    period. It the average period is within acceptable range, it will
    pulse the triac. The Sync flag is then set to allow the phase to
    be updated and the input capture is reset. If the average period
    is not within the acceptable value the input capture will be
    ignored.

*******************************************************************************/
void debugIC(void);

#pragma TRAP_PROC
void inputCapture(void)
{
    unsigned int t, pAvg;
    t = readIC();
    pAvg=calcpAvg(t);
    if (pAvg > MIN_PER && pAvg < MAX_PER)
    {
        updateDegrees();
        pulseTriac(t);
        Sync=1;
        LED=LIT;
    }
    else
    {
        LED=DIM;
    }
    resetIC();
```

```
}

/******************************************************************************

function     :   calcpAvg()

parameters   :   unsigned int t - input capture time

returns      :   unsigned int pAvg - average period

type         :   normal (called by ISR)


Description:

   This function calculates the average period over two cycles.
   The period is saved for the last two periods in an array.
   The array index Cycle is flipped after storing the current period.
   It will take two good input captures before the average period
   can be accurately calculated.


******************************************************************************/

unsigned int calcpAvg(unsigned int t)
{
     unsigned int avg;
     Period[Cycle] = t - Told;
     Told = t;
     Cycle = (Cycle + 1) & 0x01;
     avg = (Period[0] + Period[1])>>1;
     return avg;
}

/******************************************************************************

function     :   updateDegrees()

parameters   :   void

returns      :   void

type         :   normal (called by ISR)


Description:

   This function updates the base unit Degrees. The function
   is called after flipping the Cycle index. Thus, the value
   for degrees will be calculated using the previous period.
   This compensates for any waveform asymmetry. It uses a hardware
```

```
   divide function div() to avoid the slow ANSI c implementation.

*******************************************************************************/


void updateDegrees(void)
{
    Degrees = div(Period[Cycle],180);
}

/*******************************************************************************

function     :   pulseTriac()

parameters   :   unsigned int t - input capture time

returns      :   void

type         :   normal (called by ISR)


Description:

   This function will schedule the triac output pulses.
   There are four different cases. The output compares
   are scheduled differently depending on the output
   Phase.

*******************************************************************************/

void pulseTriac(unsigned int t)
{
    if (Phase <5)
    {
        noPulse();
    }
    else if (Phase <135)
    {
        shortPulse(t);
    }
    else
    {
        longPulse(t);
    }
}

/*******************************************************************************

function     :   noPulse()

parameters   :   void
```

```
returns      :   void

type         :   normal (called by ISR)


Description:

   This function does not pulse the output triac.

*******************************************************************************/

void noPulse (void)
{
}

/******************************************************************************

function     :   shortPulse()

parameters   :   unsigned int t - input capture time

returns      :   void

type         :   normal (called by ISR)


Description:

   This function generates a short pulse at the desired
   phase angle. the onTime and OffTime are calculated.
   The output compare is initializes to set the pin high
   on the next match. The OffTime is a global variable.
   The outputCompare will use the OffTime to set
   up the next edge.

*******************************************************************************/

void shortPulse (unsigned int t)
{
    unsigned int onTime;
    onTime = t + mul((180-Phase),Degrees);
    OffTime = onTime + mul(2,Degrees);
    scheduleHigh(onTime);
}

/******************************************************************************

function     :   longPulse()

parameters   :   unsigned int t - input capture time
```

```
returns      :   void

type         :   normal (called by ISR)


Description:

   This function generates a long pulse at the desired phase angle.
   The OffTime is extended out to 45 degrees. This will ensure the
   triac will not turn off prematurely, even if the current is lagging
   by up to 45 degrees.

********************************************************************************/

void longPulse (unsigned int t)
{
    unsigned int onTime, asap;
    onTime = t + mul((180-Phase),Degrees);
    OffTime = t + mul(47,Degrees);
    asap = readTCNT() + LATENCY;
    if (onTime>asap)
    {
    scheduleHigh(onTime);
    }
    else
    {
    scheduleHigh(asap);
    }
}



/*******************************************************************************

function     :   outputCompare()

parameters   :   void

returns      :   void

type         :   Interupt Service Routine


Description:

   This function is called on an output compare. If the output compare
   was set to last set the OC high then the falling edge is scheduled
   using OffTime. Otherwise the output compare function is disabled.

********************************************************************************/
```

```
#pragma TRAP_PROC
void outputCompare (void)
{
    if (OC_WAS_SET_HIGH)
    {
        scheduleLow(OffTime);
    }
    else
    {
        disableOC();
    }
}

/*******************************************************************************

function    :   initMCU()

parameters  :   void

returns     :   void

type        : low level c

description:

    This function initializes the MCU config registers, clock, ports,
    and ADC.

*******************************************************************************/

void initMCU(void)

{
    unsigned char i,j;

    CONFIG2=0x08;               /*External Clock on pin 6, Pin 7 GP I/O, rev0.7*/
   CONFIG1=0x31;             /*LVI reset disabled, LVI power disabled, COP disabled*/
    ICGMR = 0x15;               /*init ICGMR to default setting*/

    for(i=255;i!=0;i--)         /*try 256 times*/
    {
        ICGCR = 0x13;           /*switch to ext clock*/
        for(j=255;j!=0;j--);    /*wait*/
        if(ICGCR==0x13)break;   /*test*/
    }

    ADCLK=0x60;                 /*xclk / 8 for 8 MHz xtal*/

    PTA3=0;                     /*triac is off*/
    DDRA = 0x08;                /*PTA2/TCH0 is IC, PTA3/TACH1 is output */
    PTA3=0;                     /*triac is off*/
```

```
    PTB4=1;                     /* LED off (high) */
    DDRB = 0x10;                /* PTB4 LED output */
    PTB4=1;                     /* LED off (high) */
}

/*******************************************************************************

function    :   initTimer()

parameters  :   void

returns     :   void

type        : low level c

description:

    This function initializes the timer

*******************************************************************************/

void initTimer(void)
{
    TSC = 0x01;         /*TIM CLK = bus clock/2 (2.4576 MHz/2 )*/
    TMODH = 0xff;       /*set modulus register*/
    TMODL = 0xff;       /*set modulus register*/
    TSC1= PRESET_LOW;

}

/*******************************************************************************

function    :   readPot()

parameters  :   void

returns     :   unsigned char ADR

type        : low level c

description:

    This function reads the ADC ch3 and returns the value. It uses polling
    and will wait until a conversion is complete.

*******************************************************************************/
unsigned char readPot(void)
{
    ADSCR=0x03;                         /* select channel 3 */
    while((ADSCR & 0x80)==0x00);        /* wait for COCO bit */
```

```
    return (ADR);
}



/******************************************************************************
function     :   scheduleHigh()

parameters   :   unsigned int i - set high time

returns      :   void

type         :   inline assembler

description:

    This function initializes the output compare to set the pin high
    at the specified time. It uses inline assembler to ensure the
    timer control register is accessed properly.

******************************************************************************/

void scheduleHigh(unsigned int i)
{
    asm
    {
        lda TSC1_;
        ora #0x4c;  /*set CH1IE, ELSxA, ELSxB*/
        and #0x7f;  /*clear CH1F*/
        sta TSC1_;
        lda i:0;
        sta TCH1H_;
        lda i:1;
        sta TCH1L_;
    }
}

/******************************************************************************
function     :   scheduleLow()

parameters   :   unsigned int i - set low time

returns      :   void

type         :   inline assembler

description:

    This function initialize the output compare to set the pin low
    at the specified time. It uses inline assembler to ensure the
    timer control register is accessed properly.
```

```
******************************************************************************/

void scheduleLow(unsigned int i)
{
    asm
    {
        lda TSC1_;
        ora #0x48;  /* set CH1IE, set ELSxB*/
        and #0x7b;  /* clear CH1F, clear ELSxA, */
        sta TSC1_;
        lda i:0;
        sta TCH1H_;
        lda i:1;
        sta TCH1L_;
    }
}


/******************************************************************************

function    :   disableOC()

parameters  :   void

returns     :   void

type        : inline assembler


description:

    This function will disable the timer output compare function. It will
    disable further interupts and disable output compares by reading
    the high byte only.

******************************************************************************/


void disableOC(void)
{
    asm
    {
        lda TSC1_;
        and #0xb3; /clear CHIE, ELSA, ELSB*/
        sta TSC1_;
        lda TCH1H_;
    }
}

/******************************************************************************
```

```
function          :   enableIC()

parameters        :   void

returns           :   void

type              : low level c

description:

    This function will enable the timer input capture function. It uses a three
    step process to set up the timer status and control register.

********************************************************************************/

void enableIC(void)
{
    TSC0 = 0x04;        /* input capture mode, rising edges */
    TSC0 &= ~BIT7;      /*clear flag*/
    TSC0 |= BIT6;       /*enable ints*/
}

/*******************************************************************************

function          :   resetIC()

parameters        :   void

returns           :   void

type              : low level c and assembler

description:

    This function will reset the timer input capture function. It will
    access the TSCO, complement the edge trigger bits, and clear the flag.

********************************************************************************/
void resetIC()
{
    asm lda TSC0_;      /*access TSC0*/
    TSC0 ^= 0x0c;       /*flip edge*/
    TSC0 &= ~BIT7;      /*clear flag*/
}

/*******************************************************************************

function        :   readIC()

parameters      :   void
```

```
returns         :   unsigned int time (in X:A)

type            : inline assembler, no entry or exit

description:

    This function read the input capture time from the timer channel.
    The entire function is in assembler. It is necessary to read the
    timer channels in the specified order. The function returns the
    sixteen bit value.

    This function depends on the compiler returning the value in the
    X:A register. It will generate return value expected warning
    because the return value is not specified in C syntax.

    This function could also be coded as below. This version will not
    generate a warning and might work on other compilers. However, it
    will triple the code size of this function.

    unsigned int readIC(void)
    {
        unsigned int m;
        asm {
            ldx TCH0H_;
            sta m:1;
            lda TCH0L_;
            sta m:0;
            }
        return m;
    }

********************************************************************************/
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
unsigned int readIC(void)
{
    asm {
        ldx TCH0H_;
        lda TCH0L_;
        rts;
        }
}

/*******************************************************************************

function        :   readTCNT()

parameters      :   void
```

```
returns         :   unsigned int time (in X:A)

type            : inline assembler, no entry or exit

description:

    This function read the time from the timer counter.
    The entire function is in assembler. It is necessary to read the
    timer channels in the specified order. The function returns the
    sixteen bit value.

    This function depends on the compiler returning the value in the
    X:A register. It will generate return value expected warning
    because the return value is not specified in C syntax.

*******************************************************************************/
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
unsigned int readTCNT(void)
{
    asm {
        ldx TCNTH_;
        lda TCNTL_;
        rts;
        }
}

/******************************************************************************

function        :   mul()

parameters      :   unsigned char x,y (in X and A)

returns         :   unsigned int z (in X:A)

type            : inline assembler, no entry or exit

description:
    This function will multiply two 8 bit numbers together and return
    a sixteen bit value. It is coded entirely in assembler and is very
    efficient (two bytes!).

    This function is used to provide an efficient 8 x 8 multiply. An
    ANSI standard C compiler would normally promote two unsigned char
    to unsigned ints before multiplication, resulting in inefficient
    code.

*******************************************************************************/
#pragma NO_ENTRY
#pragma NO_EXIT
```

```
#pragma NO_FRAME
unsigned int mul(unsigned char x, unsigned char y)
{
    asm {
        mul;
        rts;
        }
}
/*******************************************************************************

function   :   div()

parameters :   unsigned int dvnd, unsigned char dvsr

returns    :   unsigned char dvsr (used for quotient)

type       : inline assembler, no entry or exit

description:
    This function provides a hardware 16x8 divide function.
    Standard ANSI c will promote everything to a int before
    dividing. This is much faster.

*******************************************************************************/
unsigned char div(unsigned int dvnd, unsigned char dvsr)
{
    asm {
        lda dvnd:0;
        psha;
        pulh;
        lda dvnd:1;
        ldx dvsr;
        div;
        sta dvsr;
        }
    return dvsr;
}
/*******************************************************************************

function    :   shift8()

parameters  :   unsigned int x (in X:A)

returns     :   unsigned int z (in X:A)

type        : inline assembler, no entry or exit


description:
    This function will divide a 16 bit number by 256 and return an 8
    bit number. It is coded entirely in assembler and is very efficient
```

```
    (3 bytes!).

    This function is used to provide an efficient byte shift. The
    Hiware compiler will perform a sixteen bit shift 8 times.

*******************************************************************************/
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
unsigned char shift8(unsigned int x)
{
    asm {
        txa;
        clrx;
        rts;
        }
}

/******************************************************************************

function     :   unusedVector()

parameters   :   void

returns      :   void

type         : interrupt service routine

description:
    This function provides a mechanism for unusedVectors for the HC08.

*******************************************************************************/

#pragma TRAP_PROC
void unusedVector(void)
{
}



 /*****************************************************************************
                                                  Copyright (c) Motorola 1999
File Name        :   vacuum.h

Engineer         :   Ken Berringer

Location         :   EKB

Date Created     : 1 Dec 1999
```

```
Current Revision :  1.0

Notes    :

    This header file contains constant definitions, macro definitions,
    and function prototypes for the vacuum software.

*****************************************************************************

constant definitions

*****************************************************************************/
#define ON 1
#define OFF 0
#define LED PTB4
#define LIT 0
#define DIM 1
#define MAX_PER (unsigned int)(300000/30)
#define MIN_PER (unsigned int)(300000/90)
#define UPDATE_RATE 2
#define LATENCY 48
#define UPPER_LIMIT 105
#define LOWER_LIMIT 75
#define NOTCH (UPPER_LIMIT - LOWER_LIMIT)
/*****************************************************************************

macro definitions

*****************************************************************************/


#define OC_WAS_SET_HIGH (TSC1 & 0x0c)==0x0c
#define PRESET_LOW      0x10
#define PRESET_HIGH     0x00
#define SET_ON_OC       0x1c
#define CLEAR_ON_OC     0x18
#define SET_CHIE        asm BSET 6,TSC1_
#define CLR_CHIE        asm BCLR 6,TSC1_
#define ENABLE_INTERUPTS asm cli

/*****************************************************************************

KX8 register defs

    These definitions are used for the inline assembler functions.

*****************************************************************************/

#define TCNTH_  0x21
#define TCNTL_  0x22
#define TSC0_  0x25
```

```
#define TCH0H_ 0x26
#define TCH0L_ 0x27
#define TSC1_  0x28
#define TCH1H_ 0x29
#define TCH1L_ 0x2A

/*****************************************************************************

function prototypes

*****************************************************************************/
void main(void);
void init(void);
void startupDelay(unsigned char);
unsigned char controlLoop(void);
unsigned char scale(unsigned char);
unsigned char integrate(unsigned char);
unsigned char modulate(unsigned char);
unsigned char saturate(unsigned char);
void inputCapture(void);
void updateDegrees(void);
unsigned int calcpAvg(unsigned int);
void pulseTriac(unsigned int);
void noPulse(void);
void shortPulse(unsigned int);
void longPulse(unsigned int);
void maxPulse(unsigned int);
void outputCompare(void);
void initMCU(void);
void initTimer(void);
unsigned char readPot(void);
void enableIC(void);
unsigned int readIC(void);
void resetIC(void);
void scheduleHigh(unsigned int i);
void scheduleLow(unsigned int i);
void disableOC(void);
unsigned int readTCNT(void);
unsigned int mul(unsigned char, unsigned char);
unsigned char shift8(unsigned int);
unsigned char div(unsigned int, unsigned char);

/*****************************************************************************/
```

# Section 3. Revision History

## 3.1 Contents



## 3.2 Major Changes From Application Note AN1843/D to DRM010/D Rev 0

Reformatted in new template and renamed as a Designer Reference Manual.

| Section | Page (in DRMxxx/D Rev 0) | Description of change |
|---|---|---|
| 1 | 15, 18, 19 and 20 | Addition of Figure 1-1, Figure 1-3, Figure 1-4 and Figure 1-5 |

# Revision History

# Section 4. Glossary

**A** — See “accumulators (A and B or D).”

**accumulators (A and B or D)** — Two 8-bit (A and B) or one 16-bit (D) general-purpose registers in the CPU. The CPU uses the accumulators to hold operands and results of arithmetic and logic operations.

**acquisition mode** — A mode of PLL operation with large loop bandwidth. Also see ’tracking mode’.

**address bus** — The set of wires that the CPU or DMA uses to read and write memory locations.

**addressing mode** — The way that the CPU determines the operand address for an instruction. The M68HC12 CPU has 15 addressing modes.

**ALU** — See “arithmetic logic unit (ALU).”

**analogue-to-digital converter (ATD)** — The ATD module is an 8-channel, multiplexed-input successive-approximation analog-to-digital converter.

**arithmetic logic unit (ALU)** — The portion of the CPU that contains the logic circuitry to perform arithmetic, logic, and manipulation operations on operands.

**asynchronous** — Refers to logic circuits and operations that are not synchronized by a common reference signal.

**ATD** — See “analogue-to-digital converter”.

**B** — See “accumulators (A and B or D).”

**baud rate** — The total number of bits transmitted per unit of time.

**BCD** — See “binary-coded decimal (BCD).”

**binary** — Relating to the base 2 number system.

**binary number system** — The base 2 number system, having two digits, 0 and 1. Binary arithmetic is convenient in digital circuit design because digital circuits have two permissible voltage levels, low and high. The binary digits 0 and 1 can be interpreted to correspond to the two digital voltage levels.

**binary-coded decimal (BCD)** — A notation that uses 4-bit binary numbers to represent the 10 decimal digits and that retains the same positional structure of a decimal number. For example,

234 (decimal) = 0010 0011 0100 (BCD)

**bit** — A binary digit. A bit has a value of either logic 0 or logic 1.

**branch instruction** — An instruction that causes the CPU to continue processing at a memory location other than the next sequential address.

**break module** — The break module allows software to halt program execution at a programmable point in order to enter a background routine.

**breakpoint** — A number written into the break address registers of the break module. When a number appears on the internal address bus that is the same as the number in the break address registers, the CPU executes the software interrupt instruction (SWI).

**break interrupt** — A software interrupt caused by the appearance on the internal address bus of the same value that is written in the break address registers.

**bus** — A set of wires that transfers logic signals.

**bus clock** — See "CPU clock".

**byte** — A set of eight bits.

**CAN** — See "Motorola scalable CAN."

**CCR** — See “condition code register.”

**central processor unit (CPU)** — The primary functioning unit of any computer system. The CPU controls the execution of instructions.

**CGM** — See “clock generator module (CGM).”

**clear** — To change a bit from logic 1 to logic 0; the opposite of set.

**clock** — A square wave signal used to synchronize events in a computer.

**clock generator module (CGM)** — The CGM module generates a base clock signal from which the system clocks are derived. The CGM may include a crystal oscillator circuit and/or phase-locked loop (PLL) circuit.

**comparator** — A device that compares the magnitude of two inputs. A digital comparator defines the equality or relative differences between two binary numbers.

**computer operating properly module (COP)** — A counter module that resets the MCU if allowed to overflow.

**condition code register (CCR)** — An 8-bit register in the CPU that contains the interrupt mask bit and five bits that indicate the results of the instruction just executed.

**control bit** — One bit of a register manipulated by software to control the operation of the module.

**control unit** — One of two major units of the CPU. The control unit contains logic functions that synchronize the machine and direct various operations. The control unit decodes instructions and generates the internal control signals that perform the requested operations. The outputs of the control unit drive the execution unit, which contains the arithmetic logic unit (ALU), CPU registers, and bus interface.

**COP** — See "computer operating properly module (COP)."

**CPU** — See "central processor unit (CPU)."

**CPU12** — The CPU of the MC68HC12 Family.

**CPU clock** — Bus clock select bits BCSP and BCSS in the clock select register (CLKSEL) determine which clock drives SYSCLK for the main system, including the CPU and buses. When EXTALi drives the SYSCLK, the CPU or bus clock frequency ($f_o$) is equal to the EXTALi frequency divided by 2.

**CPU cycles** — A CPU cycle is one period of the internal bus clock, normally derived by dividing a crystal oscillator source by two or more so the high and low times will be equal. The length of time required to execute an instruction is measured in CPU clock cycles.

**CPU registers** — Memory locations that are wired directly into the CPU logic instead of being part of the addressable memory map. The CPU always has direct access to the information in these registers. The CPU registers in an M68HC12 are:

- A (8-bit accumulator)
- B (8-bit accumulator)
  - D (16-bit accumulator formed by concatenation of accumulators A and B)
- IX (16-bit index register)
- IY (16-bit index register)

- SP (16-bit stack pointer)
- PC (16-bit program counter)
- CCR (8-bit condition code register)

**cycle time** — The period of the operating frequency: $t_{CYC} = 1/f_{OP}$.

**D** — See “accumulators (A and B or D).”

**decimal number system** — Base 10 numbering system that uses the digits zero through nine.

**duty cycle** — A ratio of the amount of time the signal is on versus the time it is off. Duty cycle is usually represented by a percentage.

**ECT** — See “enhanced capture timer.”

**EEPROM** — Electrically erasable, programmable, read-only memory. A nonvolatile type of memory that can be electrically erased and reprogrammed.

**EPROM** — Erasable, programmable, read-only memory. A nonvolatile type of memory that can be erased by exposure to an ultraviolet light source and then reprogrammed.

**enhanced capture timer (ECT)** — The HC12 Enhanced Capture Timer module has the features of the HC12 Standard Timer module enhanced by additional features in order to enlarge the field of applications.

**exception** — An event such as an interrupt or a reset that stops the sequential execution of the instructions in the main program.

**fetch** — To copy data from a memory location into the accumulator.

**firmware** — Instructions and data programmed into nonvolatile memory.

**free-running counter** — A device that counts from zero to a predetermined number, then rolls over to zero and begins counting again.

**full-duplex transmission** — Communication on a channel in which data can be sent and received simultaneously.

**hexadecimal** — Base 16 numbering system that uses the digits 0 through 9 and the letters A through F.

**high byte** — The most significant eight bits of a word.

**illegal address** — An address not within the memory map

**illegal opcode** — A nonexistent opcode.

**index registers (IX and IY)** — Two 16-bit registers in the CPU. In the indexed addressing modes, the CPU uses the contents of IX or IY to determine the effective address of the operand. IX and IY can also serve as a temporary data storage locations.

**input/output (I/O)** — Input/output interfaces between a computer system and the external world. A CPU reads an input to sense the level of an external signal and writes to an output to change the level on an external signal.

**instructions** — Operations that a CPU can perform. Instructions are expressed by programmers as assembly language mnemonics. A CPU interprets an opcode and its associated operand(s) and instruction.

**inter-IC bus ($I^2C$) — A** two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange between devices.

**interrupt** — A temporary break in the sequential execution of a program to respond to signals from peripheral devices by executing a subroutine.

**interrupt request** — A signal from a peripheral to the CPU intended to cause the CPU to execute a subroutine.

**I/O** — See "input/output (I/0)."

**jitter** — Short-term signal instability.

**latch** — A circuit that retains the voltage level (logic 1 or logic 0) written to it for as long as power is applied to the circuit.

**latency** — The time lag between instruction completion and data movement.

**least significant bit (LSB)** — The rightmost digit of a binary number.

**logic 1** — A voltage level approximately equal to the input power voltage ($V_{DD}$).

**logic 0** — A voltage level approximately equal to the ground voltage ($V_{SS}$).

**low byte** — The least significant eight bits of a word.

**M68HC12** — A Motorola family of 16-bit MCUs.

**mark/space** — The logic 1/logic 0 convention used in formatting data in serial communication.

**mask** — 1. A logic circuit that forces a bit or group of bits to a desired state. 2. A photomask used in integrated circuit fabrication to transfer an image onto silicon.

**MCU** — Microcontroller unit. See "microcontroller."

**memory location** — Each M68HC12 memory location holds one byte of data and has a unique address. To store information in a memory location, the CPU places the address of the location on the address bus, the data information on the data bus, and asserts the write signal. To read information from a memory location, the CPU places the address of the location on the address bus and asserts the read signal. In response to the read signal, the selected memory location places its data onto the data bus.

**memory map** — A pictorial representation of all memory locations in a computer system.

**MI-Bus —** See "Motorola interconnect bus".

**microcontroller** — Microcontroller unit (MCU). A complete computer system, including a CPU, memory, a clock oscillator, and input/output (I/O) on a single integrated circuit.

**modulo counter** — A counter that can be programmed to count to any number from zero to its maximum possible modulus.

**most significant bit (MSB)** — The leftmost digit of a binary number.

**Motorola interconnect bus (MI-Bus)** — The Motorola Interconnect Bus (MI Bus) is a serial communications protocol which supports distributed real-time control efficiently and with a high degree of noise immunity.

**Motorola scalable CAN (msCAN) —** The Motorola scalable controller area network is a serial communications protocol that efficiently supports distributed real-time control with a very high level of data integrity.

**msCAN —** See "Motorola scalable CAN".

**MSI —** See "multiple serial interface".

**multiple serial interface —** A module consisting of multiple independent serial I/O sub-systems, e.g. two SCI and one SPI.

**multiplexer** — A device that can select one of a number of inputs and pass the logic level of that input on to the output.

**nibble** — A set of four bits (half of a byte).

**object code** — The output from an assembler or compiler that is itself executable machine code, or is suitable for processing to produce executable machine code.

**opcode** — A binary code that instructs the CPU to perform an operation.

**open-drain** — An output that has no pullup transistor. An external pullup device can be connected to the power supply to provide the logic 1 output voltage.

**operand** — Data on which an operation is performed. Usually a statement consists of an operator and an operand. For example, the operator may be an add instruction, and the operand may be the quantity to be added.

**oscillator** — A circuit that produces a constant frequency square wave that is used by the computer as a timing and sequencing reference.

**OTPROM** — One-time programmable read-only memory. A nonvolatile type of memory that cannot be reprogrammed.

**overflow** — A quantity that is too large to be contained in one byte or one word.

**page zero** — The first 256 bytes of memory (addresses $0000–$00FF).

**parity** — An error-checking scheme that counts the number of logic 1s in each byte transmitted. In a system that uses odd parity, every byte is expected to have an odd number of logic 1s. In an even parity system, every byte should have an even number of logic 1s. In the transmitter, a parity generator appends an extra bit to each byte to make the number of logic 1s odd for odd parity or even for even parity. A parity checker in the receiver counts the number of logic 1s in each byte. The parity checker generates an error signal if it finds a byte with an incorrect number of logic 1s.

**PC** — See “program counter (PC).”

**peripheral** — A circuit not under direct CPU control.

**phase-locked loop (PLL)** — A clock generator circuit in which a voltage controlled oscillator produces an oscillation which is synchronized to a reference signal.

**PLL** — See "phase-locked loop (PLL)."

**pointer** — Pointer register. An index register is sometimes called a pointer register because its contents are used in the calculation of the address of an operand, and therefore points to the operand.

**polarity** — The two opposite logic levels, logic 1 and logic 0, which correspond to two different voltage levels, $V_{DD}$ and $V_{SS}$.

**polling** — Periodically reading a status bit to monitor the condition of a peripheral device.

**port** — A set of wires for communicating with off-chip devices.

**prescaler** — A circuit that generates an output signal related to the input signal by a fractional scale factor such as 1/2, 1/8, 1/10 etc.

**program** — A set of computer instructions that cause a computer to perform a desired operation or operations.

**program counter (PC)** — A 16-bit register in the CPU. The PC register holds the address of the next instruction or operand that the CPU will use.

**pull** — An instruction that copies into the accumulator the contents of a stack RAM location. The stack RAM address is in the stack pointer.

**pullup** — A transistor in the output of a logic gate that connects the output to the logic 1 voltage of the power supply.

**pulse-width** — The amount of time a signal is on as opposed to being in its off state.

**pulse-width modulation (PWM)** — Controlled variation (modulation) of the pulse width of a signal with a constant frequency.

**push** — An instruction that copies the contents of the accumulator to the stack RAM. The stack RAM address is in the stack pointer.

**PWM period** — The time required for one complete cycle of a PWM waveform.

**RAM** — Random access memory. All RAM locations can be read or written by the CPU. The contents of a RAM memory location remain valid until the CPU writes a different value or until power is turned off.

**RC circuit** — A circuit consisting of capacitors and resistors having a defined time constant.

**read** — To copy the contents of a memory location to the accumulator.

**register** — A circuit that stores a group of bits.

**reserved memory location** — A memory location that is used only in special factory test modes. Writing to a reserved location has no effect. Reading a reserved location returns an unpredictable value.

**reset** — To force a device to a known condition.

**SCI** — See "serial communication interface module (SCI)."

**serial** — Pertaining to sequential transmission over a single line.

**serial communications interface module (SCI)** — A module that supports asynchronous communication.

**serial peripheral interface module (SPI)** — A module that supports synchronous communication.

**set** — To change a bit from logic 0 to logic 1; opposite of clear.

**shift register** — A chain of circuits that can retain the logic levels (logic 1 or logic 0) written to them and that can shift the logic levels to the right or left through adjacent circuits in the chain.

**signed** — A binary number notation that accommodates both positive and negative numbers. The most significant bit is used to indicate whether the number is positive or negative, normally logic 0 for positive and logic 1 for negative. The other seven bits indicate the magnitude of the number.

**software** — Instructions and data that control the operation of a microcontroller.

**software interrupt (SWI)** — An instruction that causes an interrupt and its associated vector fetch.

**SPI** — See "serial peripheral interface module (SPI)."

**stack** — A portion of RAM reserved for storage of CPU register contents and subroutine return addresses.

**stack pointer (SP)** — A 16-bit register in the CPU containing the address of the next available storage location on the stack.

**start bit** — A bit that signals the beginning of an asynchronous serial transmission.

**status bit** — A register bit that indicates the condition of a device.

**stop bit** — A bit that signals the end of an asynchronous serial transmission.

**subroutine** — A sequence of instructions to be used more than once in the course of a program. The last instruction in a subroutine is a return from subroutine (RTS) instruction. At each place in the main program where the subroutine instructions are needed, a jump or branch to subroutine (JSR or BSR) instruction is used to call the subroutine. The CPU leaves the flow of the main program to execute the instructions in the subroutine. When the RTS instruction is executed, the CPU returns to the main program where it left off.

**synchronous** — Refers to logic circuits and operations that are synchronized by a common reference signal.

**timer** — A module used to relate events in a system to a point in time.

**toggle** — To change the state of an output from a logic 0 to a logic 1 or from a logic 1 to a logic 0.

**tracking mode** — A mode of PLL operation with narrow loop bandwidth. Also see 'acquisition mode.'

**two's complement** — A means of performing binary subtraction using addition techniques. The most significant bit of a two's complement number indicates the sign of the number (1 indicates negative). The two's complement negative of a number is obtained by inverting each bit in the number and then adding 1 to the result.

**unbuffered** — Utilizes only one register for data; new data overwrites current data.

**unimplemented memory location** — A memory location that is not used. Writing to an unimplemented location has no effect. Reading an unimplemented location returns an unpredictable value.

**variable** — A value that changes during the course of program execution.

**VCO** — See "voltage-controlled oscillator."

**vector** — A memory location that contains the address of the beginning of a subroutine written to service an interrupt or reset.

**voltage-controlled oscillator (VCO)** — A circuit that produces an oscillating output signal of a frequency that is controlled by a dc voltage applied to a control input.

**waveform** — A graphical representation in which the amplitude of a wave is plotted against time.

**wired-OR** — Connection of circuit outputs so that if any output is high, the connection point is high.

**word** — A set of two bytes (16 bits).

**write** — The transfer of a byte of data from the CPU to a memory location.

DRM010/D

**HOW TO REACH US:**

**USA/EUROPE/LOCATIONS NOT LISTED:**

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

**JAPAN:**

Motorola Japan Ltd.; SPS, Technical Information Center,
3-20-1, Minami-Azabu Minato-ku, Tokyo 106-8573 Japan
81-3-3440-3569

**ASIA/PACIFIC:**

Motorola Semiconductors H.K. Ltd.;
Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

**TECHNICAL INFORMATION CENTER:**

1-800-521-6274

HOME PAGE:

http://motorola.com/semiconductors



MOTOROLA