Application Note

Z8 Encore! XP® F1680 Microstepping Controller

AN027202-0312

Abstract

Zilog's Z8 Encore! XP® F1680 Series incorporates a best-in-class feature set that is optimized for *stepper motor microstepping* controls. The features of Z8 Encore! XP F1680 include:

- A fast 11 MHz internal oscillator
- Two analog comparators
- 10-bit Analog-to-Digital Converter (ADC)
- Multichannel PWM timer
- Three general-purpose timers

This Application Note describes how to drive a unipolar stepper motor using the Z8 Encore! XP F1680 MCU's onboard analog comparators for one-shot feedback current limiting. It also describes the F1680's multichannel timer as a microstepper sine/cosine current generator.

Notes: The source code file associated with this application note, AN0272-SC01.zip, is available for download on zilog.com. This source code has been tested with version 5.0.0 of ZDS II for Z8 Encore! XP-powered MCUs. Subsequent releases of ZDS II may require you to modify the code supplied with this application note.

Theory of Operation

Microstepping, or sine/cosine microstepping, is a stepper motor drive technique in which the current in the motor windings is controlled to approximate a sinusoidal waveform. Microstepping produces a much smoother rotation than that of a full step drive, plus it provides greater resolution and freedom from resonance problems because it involves more steps per revolution.

In a conventional full step drive, an equal amount of current is applied to each of a motor's stator coils. The magnetic rotor aligns itself in the coil's magnetic field. With each motor step, current is reversed in one of the coils and the rotor realigns to the new magnetic field to move the rotor one motor step, i.e., 90 degrees. See Figure 1.

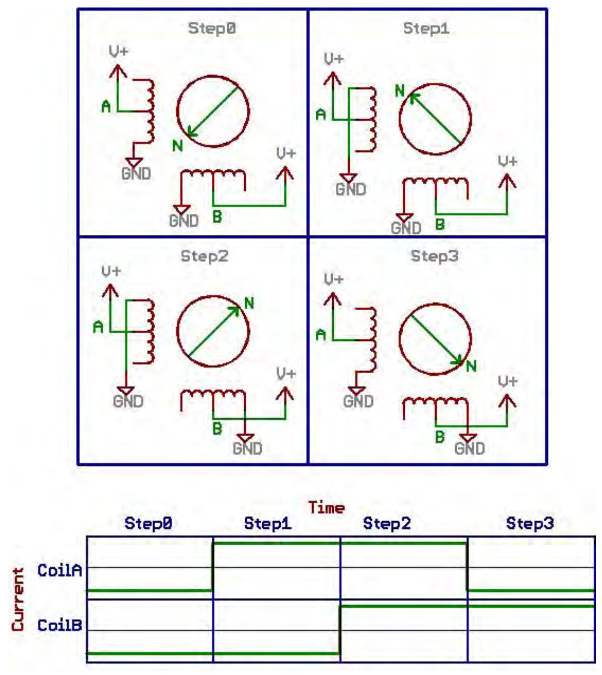


Figure 1. Microstepping

With microstepping, varying amounts of current are applied to a motor's coils so that the magnetic field smoothly transitions from one polarity to the next. Each full step is divided into several microsteps of varying current to produce a larger number of magnetic fields that the rotor can align with. The result is smoother motor rotation, quieter operation and greater motor resolution. See Figure 2.

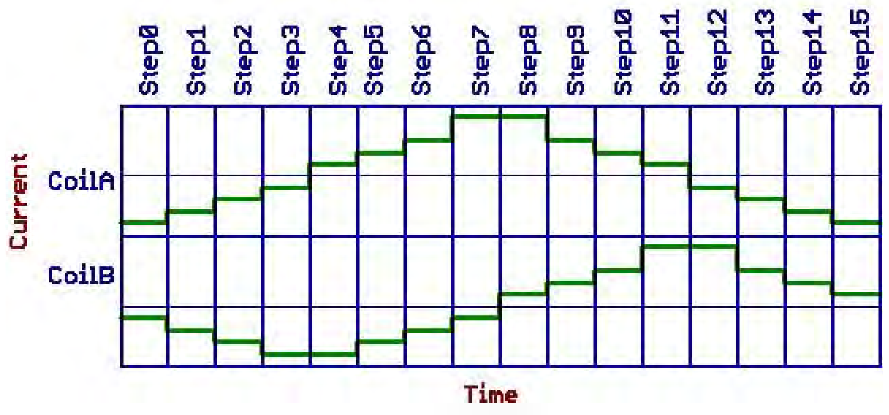


Figure 2. Microsteps of Varying Current

Hardware Architecture

To proceed from theory to demonstration, we developed an application for a motor that requires two current generators, one for each coil. This application includes the following elements:

- A potentiometer for adjusting motor speed
- A switch to turn the motor ON and OFF
- A switch to reverse the direction of the motor
- A switch to advance the motor one step

User Interface

Switches SW1, SW2 and SW3 are pulled down when pressed, and are pulled up by the F1680 MCU's internal pull-ups. Speed potentiometer R1 is read by the F1680 MCU's internal Successive Approximation Register (SAR) ADC, using V_{DD} as a reference.

Current Generator

The motor's two current generators use several of the Z8 Encore! XP F1680 MCU's features. Figure 3 displays one of the two current generators.

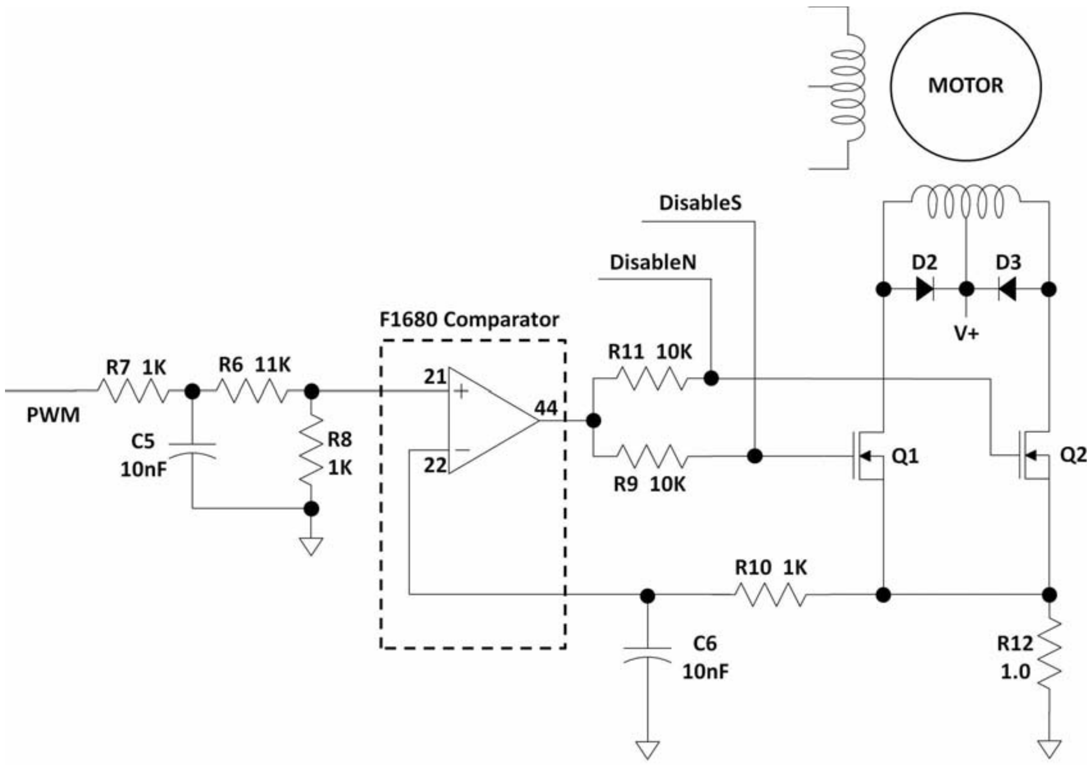


Figure 3. Current Generator

The F1680 MCU provides a PWM signal that is averaged by R7 and C5. The PWM duty cycle represents the desired current to be produced by the generator. R6 and R8 attenuate the averaged PWM signal so that the PWM signal is scaled properly for the noninverting input of the F1680's comparator. R12 is a current sense resistor for the motor's winding, while R10 and C6 provide a small amount of delay for the comparator. When Q1 or Q2 is turned ON, current starts building in the motor winding, and the voltage drop across R12 also builds until the F1680 MCU's comparator trips at the desired motor current. In some applications, an additional diode may be required to protect the body diode of each of the field effect transistors (FETs) from excessive dissipation. A simulation of this process is shown in Figure 4.

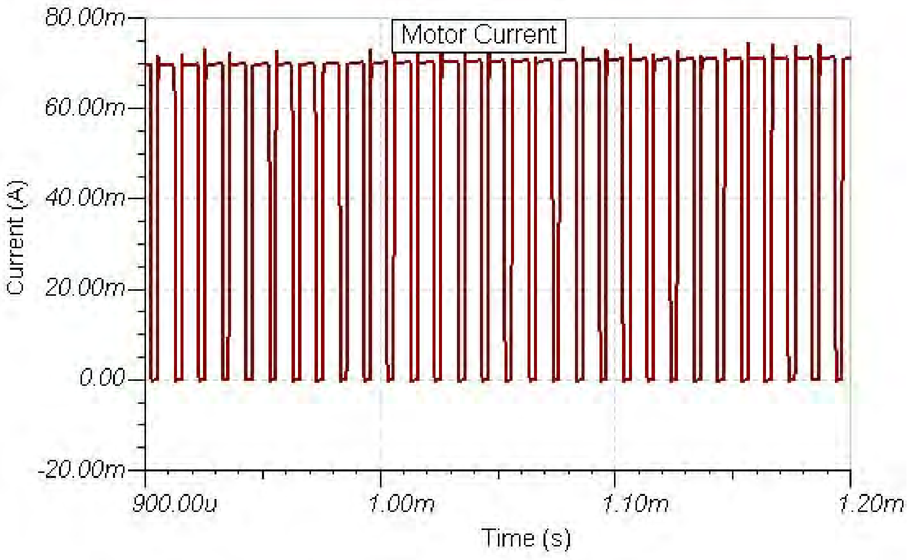


Figure 4. Simulation

The DisableS and DisableN signals determine the coil's polarity. These I/O pins are configured as open-drain. Therefore, when the port pin is High, the appropriate transistor is turned ON with the PWM signal from the comparator, whereas when the port pin is pulled Low, it turns OFF the coil by pulling the transistor's gate Low.

Multichannel Timer

The F1680 MCU's multichannel timer provides two PWMs required for the current generator's reference. The multichannel timer has a 16-bit up/down counter with a prescaler and four independent capture/compare channels that can be used for ONE-SHOT, CONTINUOUS, PWM or CAPTURE modes. In PWM output operation, the channel generates a PWM output signal on the channel output pin (TOutA, B, C or D). The channel output toggles whenever the timer count matches the channel compare value in the match registers (MCTCHyH and MCTCHyL) and when the count terminates. See Figure 5.

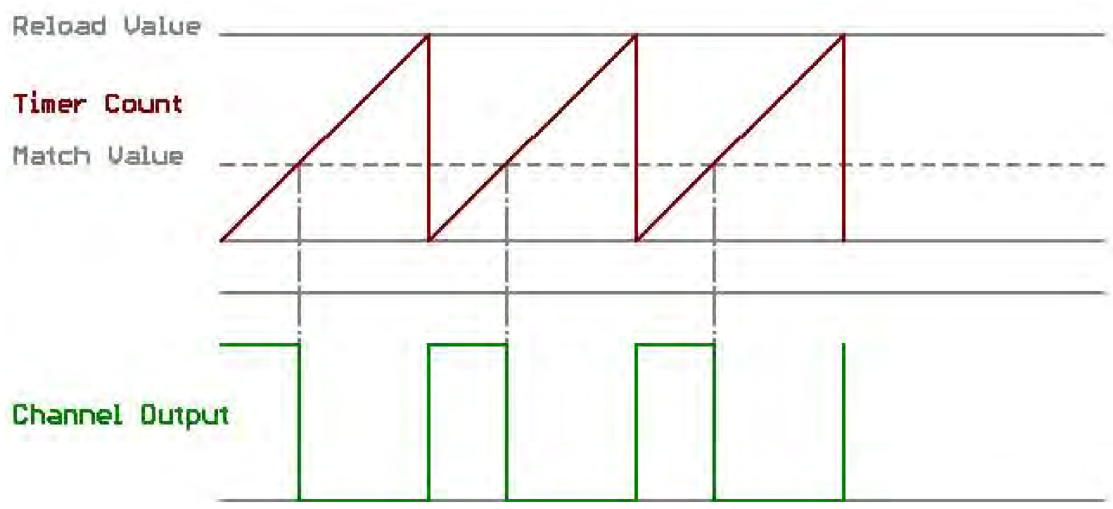


Figure 5. Multichannel Timer

The MCTCHyH and MCTCHyL match registers are buffered and are written to at any time without corrupting the PWM. Any changes to the registers are delayed until the next timer end count.

Comparator

Z8 Encore! XP F1680 MCU features two onboard comparators with programmable internal references. In this application the comparators are used as freestanding comparators that free the design from an additional external hardware.

Open-Drain Outputs

An additional feature of the Z8 Encore! Series microcontrollers is the ability for the ports to be configured as open-drain. This means that the output port pin will not source any current when it is High but will sink current when it is Low. The ability to use port pins as open-drain helps to solve awkward design problems that would otherwise require additional external hardware.

Software Implementation

The application requires the configuration of timers, ports and interrupts, as described in this section.

IdleTimers()

The `IdleTimers()` function configures the following three timers, which are required to run the microstepper application.

Timer0. Timer0 runs in CONTINUOUS Mode and is used for main loop timing with a frequency of 60 Hz.

Timer1. Timer1 executes in CONTINUOUS Mode; its frequency is varied for use with the motor's speed control. This timer triggers the interrupt service routine (ISR) that changes the current which is applied to the motor's windings.

Multichannel Timer. The multichannel timer is used to generate the two PWM signals used for the motor's current regulators.

IdlePorts()

The `IdlePorts()` function configures the ports' direction, pull-ups, and any alternate functions that the ports require.

RefreshPorts()

Similar to `IdlePorts()`, the `RefreshPorts()` function also configures the ports' direction, pull-ups and any alternate functions. In noisy environments, port directions can be corrupted; therefore, it is preferred that the port direction register is refreshed regularly.

MotorISR()

The actual driving of the motor is performed within a single interrupt function which services the motor's current generators and coil enablers. The `MotorISR()` function is the Timer1 ISR and is serviced when Timer1 expires, or when requested manually by pressing the Step key.

This `MotorISR()` function first reloads Timer1's reload register with the current speed. Because this interrupt is serviced when Timer1 expires, Timer1 is close to zero and is safely reloaded with a new value. If the timer is reloaded with a value lower than its current value, then the timer continues to count up until it overflows past `0xFFFF`. It then returns to the value that is currently loaded in the reload register. This overflow momentarily stalls the motor; in effect, the timer must be loaded with its new value at the start of the ISR.

Next, this function uses the `motorFwd` variable to determine the direction in which the motor is turned; i.e., forward or backward. The forward direction increments the pointers used for driving the motor steps; a reverse direction decrements the pointers. There are three pointers: two for the current generators and one for the coil enablers. `CoilPtr` points to the ROM table `Coils[]`, which defines the coil enablers for each full step. `CoilPtr` is changed whenever the polarity of a coil changes. PhaseAPtr and PhaseBPtr point at ROM tables `PhaseA[]` and `PhaseB[]`, which each contain timer values corresponding to each microstep. These `PhaseA[]` and `PhaseB[]` table values are selected to produce a sine and cosine current function for the motor's coils; these values are loaded into the multichannel Timer Match registers to generate a PWM signal that is averaged and used as the comparator's reference. Including interrupt latency, the entire ISR requires 23.5 µs (minimum) to execute, allowing for fine microsteps and a high pulse-per-minute speed.

Main Loop

The Main loop cycles at 60 Hz and performs all of the tasks required for the application. First, all of the port modes are refreshed to insure that they are always in the correct direction, pull-ups are enabled, alternate functions are selected, open-drain pins are correctly configured, and interrupts are enabled. Next, the functions for reading the keyboard and pot are called, and then the User Interface function is called. The Main loop then waits for the 60 Hz timer to expire before repeating.

DebounceKeyboard()

The `DebounceKeyboard()` function is called once during every Main loop to read and debounce the keyboard. This function is highly flexible and can be used in a wide variety of applications. The function first calls the `ReadKeyboard()` function to return the status of any key that is pressed. Before considering any keypress valid, the keyboard must be enabled by debouncing the condition of all open keys. After the keyboard is enabled, the same key must be closed for a number of Main loop cycles before it is considered a valid keypress. After the keypress is considered valid, the keyboard is disabled to prevent the system from seeing repeat keypresses. The keyboard is not enabled again until all keys have been debounced in the open condition again. Certain keys are allowed to repeat if they are held closed; in this application, the Step key repeats if it is held down. The speed of the repeat, or *slewing*, has two rates: when the key is first closed it repeats at a slow rate for a number of cycles and then speeds up and repeats at a much higher rate.

UserInterface()

The `UserInterface()` function is called once per Main loop. This function acts upon the keys that are pressed and determines if the motor should turn ON, turn OFF, change speed, change direction, or step.

ReadKeyboard()

The `ReadKeyboard()` function is called once per Main loop by the `DebounceKeyboard()` function. This function performs the keyboard read at the I/O level.

ReadPot()

The `ReadPot()` function is called once per Main loop by the `DebounceKeyboard()` function. This function uses the ADC to read the position of the speed potentiometer.

Wait()

The `Wait()` function polls the Timer0 interrupt flag and pauses execution until the timer trips the flag. The interrupt flag is reset again before exiting.

Testing

A circuit board is built (see the schematic diagram in Schematics on page 10), and a Nippon PF35T-48L4 motor is used as the load. See Figure 6.

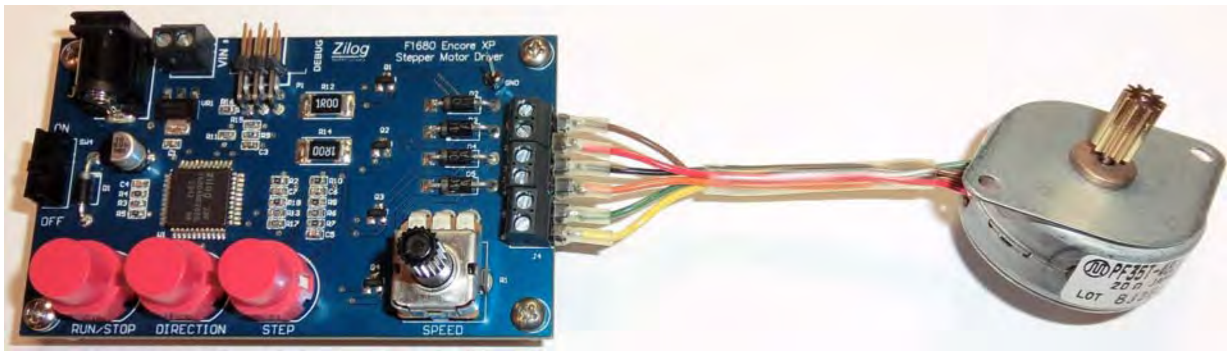


Figure 6. Printed Circuit Board Connected to Motor

The motor is tested from a minimum speed of 19 pulses per second to a speed of 190 pulses per second in both directions, at full speed and single-stepped. The coil current is measured for proper operation by reading the voltage drop across R12 and R14 with an oscilloscope; see Figure 7.

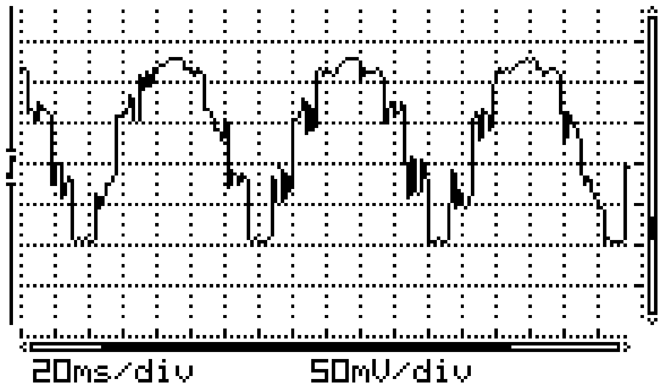


Figure 7. Coil Current

The execution time of the Main loop is measured at 47 µs; the execution time of the ISR is measured at 21 µs. The maximum software microstep speed in this application is calculated using the following equation:

$$\frac{1\text{sec} - (47\mu\text{s}*60)}{21\mu\text{s}} = 47485 \text{ microsteps per second}$$

Modifications

A motor other than the Nippon PF35T-48L4 motor used in this application may require a different drive voltage and current. Changing the value of sense resistors R12 and R14 changes the motor current. Larger motors may also require a change to the Q1–Q4 values or additional diodes to protect the body diodes of these transistors.

Additional Microsteps

Additional microsteps can be added by changing the definition of `MicroSteps` in the `StepperIO.h` file and by inserting additional entries into the `PhaseA[]` and `PhaseB[]` tables.

Compensated Sine/Cosine Profiles

As written, the software generates ideal sine/cosine current profiles for the motor. However, an ideal current profile does not guarantee the best step accuracy. Because of a varying air gap area, air gap distance, and/or magnetic hysteresis, flux vector direction and magnitude can deviate from the ideal sine/cosine behavior in the motor. Accuracy can be improved for a particular motor by altering the entries in the `PhaseA[]` and `PhaseB[]` tables to tailor the sine/cosine profile to the motor.

Summary

The Z8 Encore! XP® F1680 Series MCUs have an ideal set of peripherals for low-cost microstepping motor control. The F1680's onboard comparators can be configured for one-shot feedback current limiting, while the multichannel timer provides a PWM reference for a sine/cosine current generator. The compact solution developed for the purpose of this application note requires only 12 I/O pins and a single interrupt for all microstepping functions.

References

The following documentation supports this application note and/or further describes the Z8 Encore! XP F1680 Series MCU.

- Z8 Encore! XP F1680 Series Product Specification (PS0250)
- eZ8 CPU User Manual (UM0128)
- Zilog Developer Studio II – Z8 Encore! User Manual (UM0130)
- Control of Stepper Motors (http://www.cs.uiowa.edu/~jones/step/)

Appendix A. Schematics

Figure 8 displays the schematic for the Z8 Encore! XP F1680 MCU-based microstepping controller.

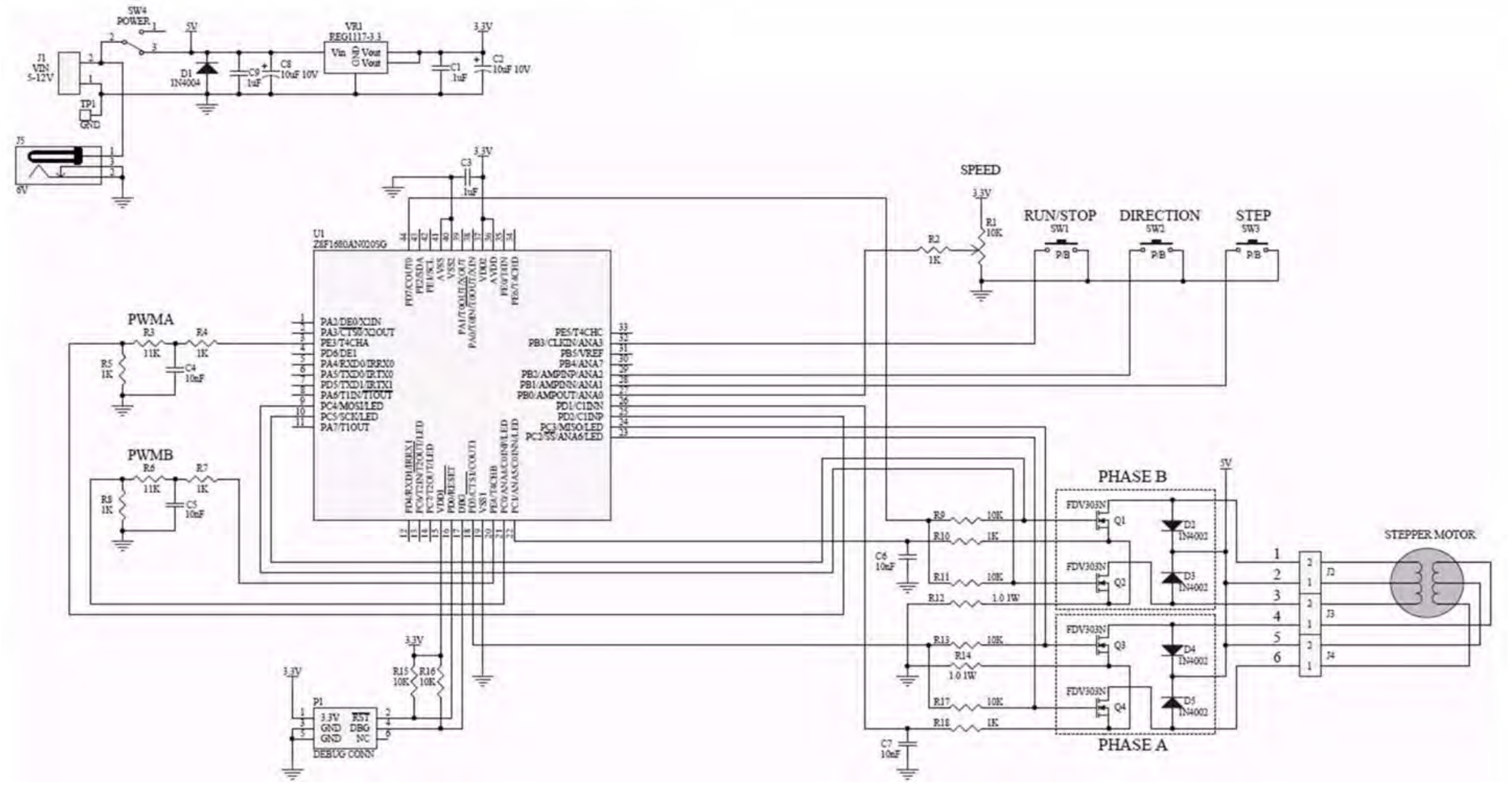
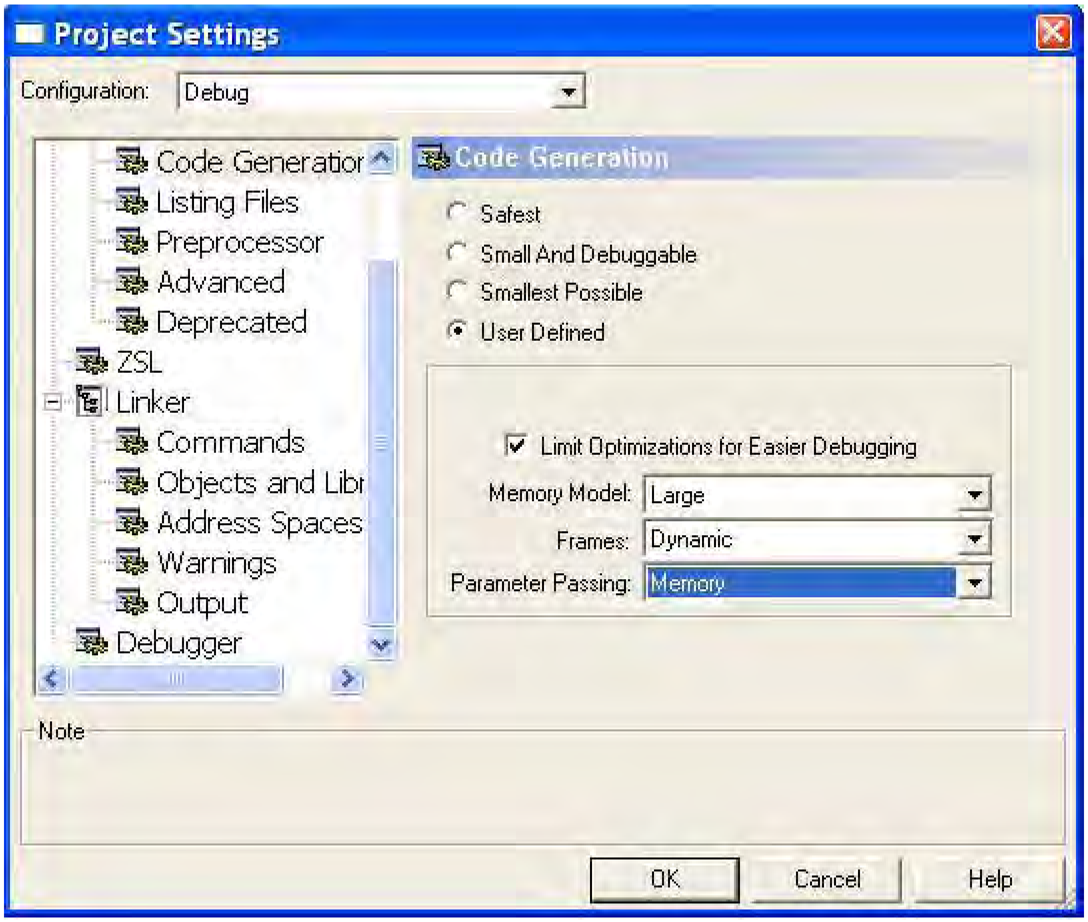


Figure 8. Schematic for Z8 Encore! XP F1680 Microstepping Controller

Appendix B. Project Settings in ZDSII

Figure 9 shows the required debug configuration within the ZDSII Project Settings dialog for the source code associated with this application note.


Project Settings

Configuration: Debug

Code Generation
- ( ) Safest
- ( ) Small And Debuggable
- ( ) Smallest Possible
- (●) User Defined

☑ Limit Optimizations for Easier Debugging

Memory Model: Large

Frames: Dynamic

Parameter Passing: Memory

Figure 9. Microstepper Motor Debug Configuration

Appendix C. Flowcharts

Figures 10 through 16 display the flowcharts for the API functions.

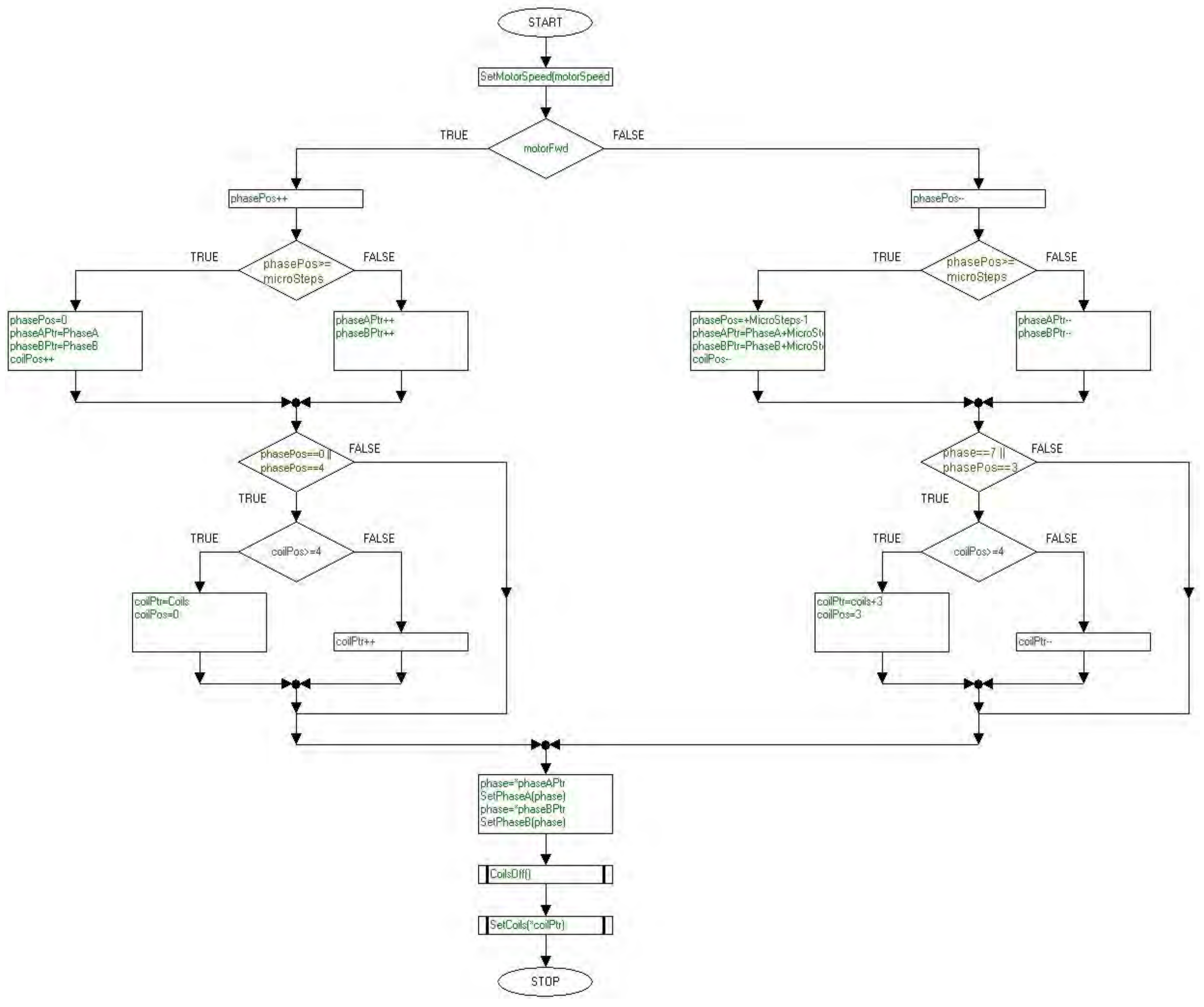


Figure 10. MotorISR() Function Flow

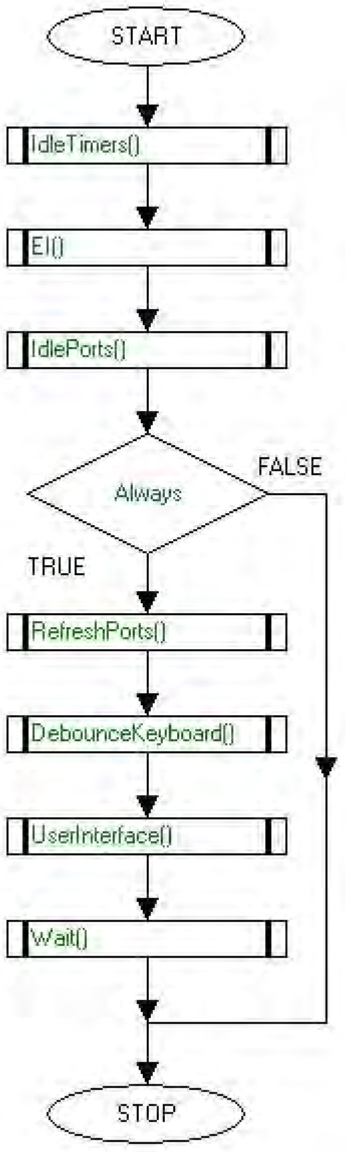


Figure 11. Main() Function Flow

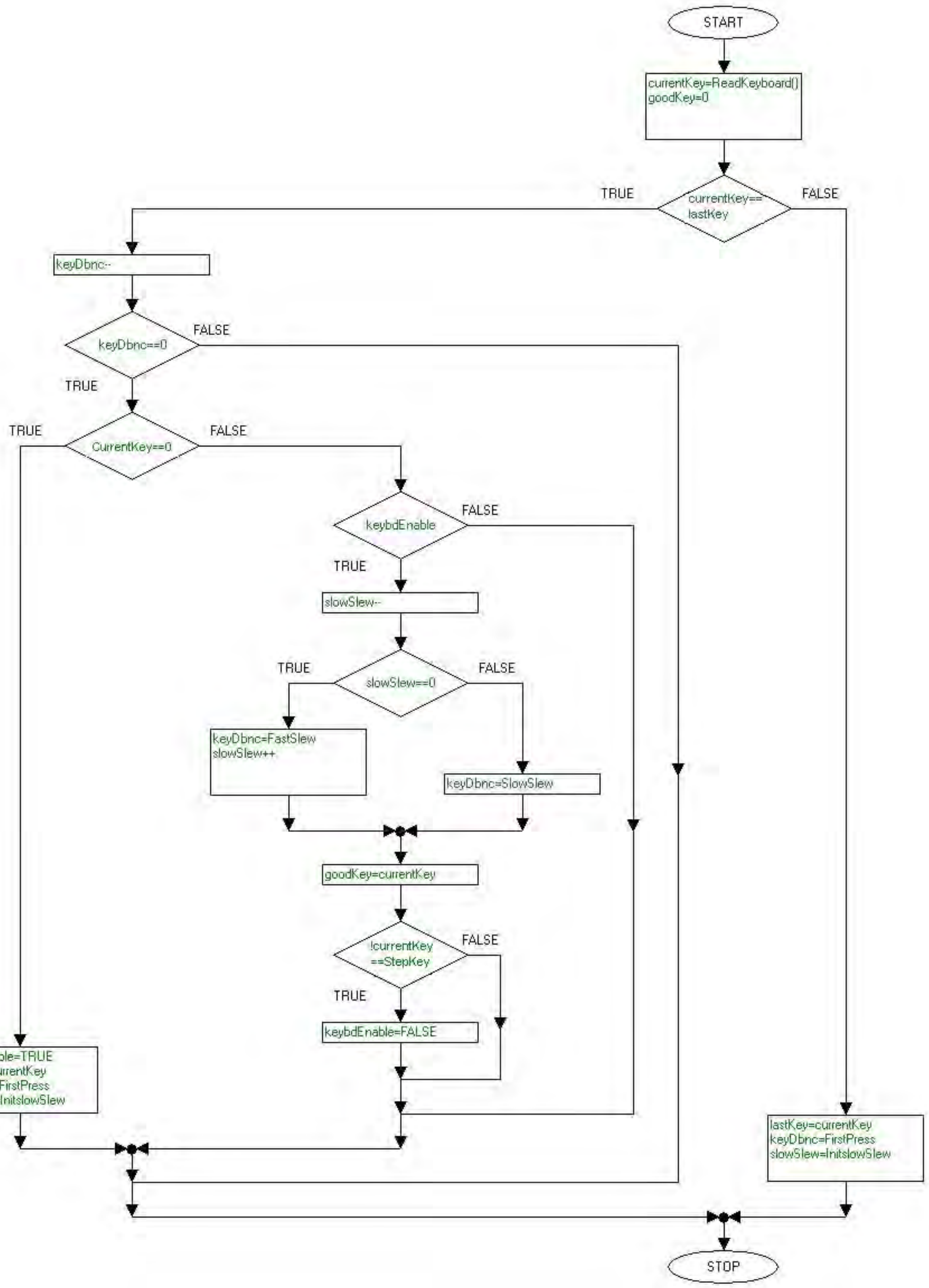


Figure 12. DebounceKeyboard() Function Flow

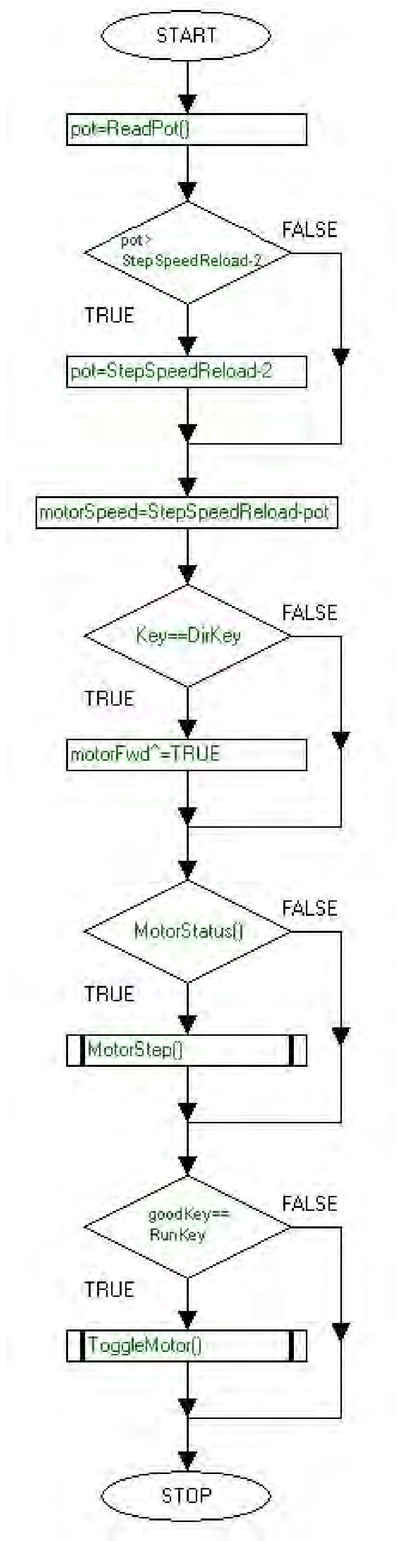


Figure 13. UserInterface() Function Flow

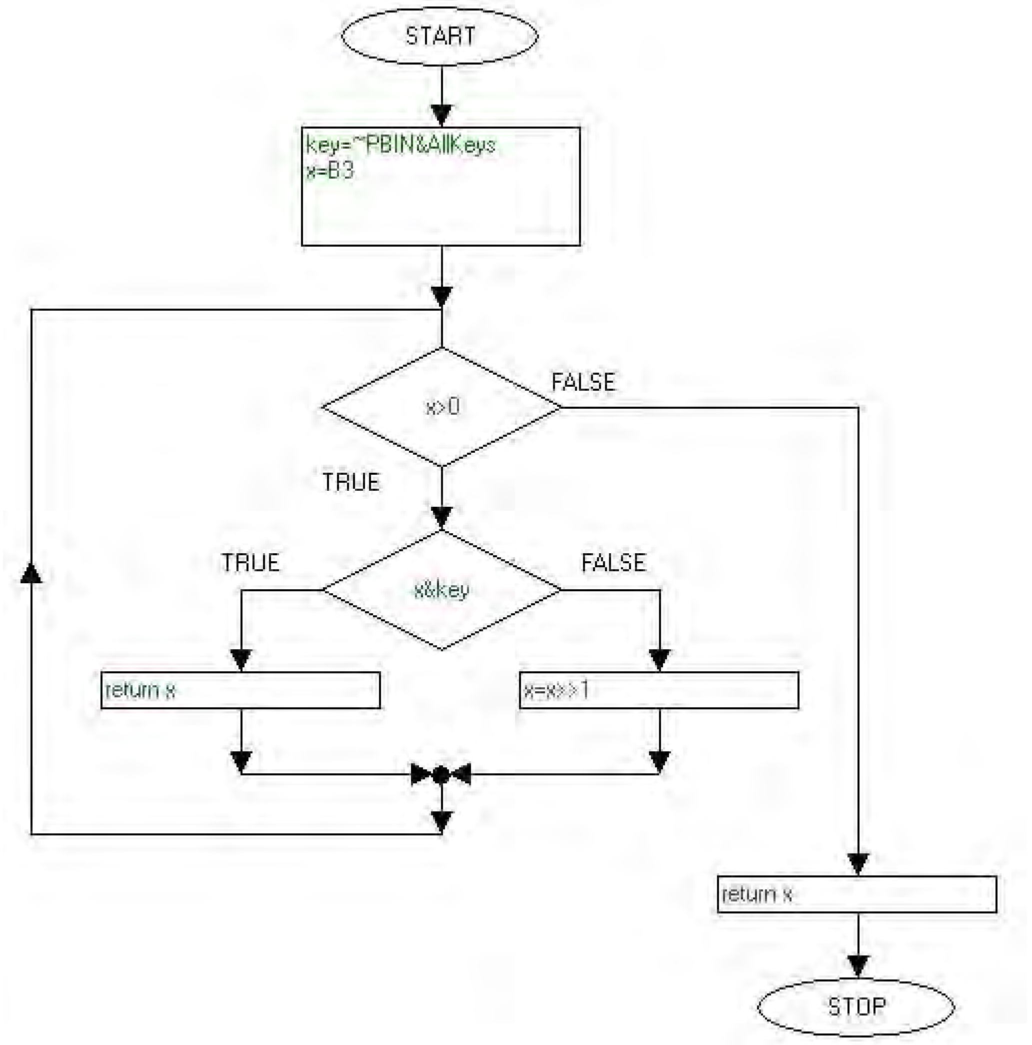


Figure 14. ReadKeyboard() Function Flow

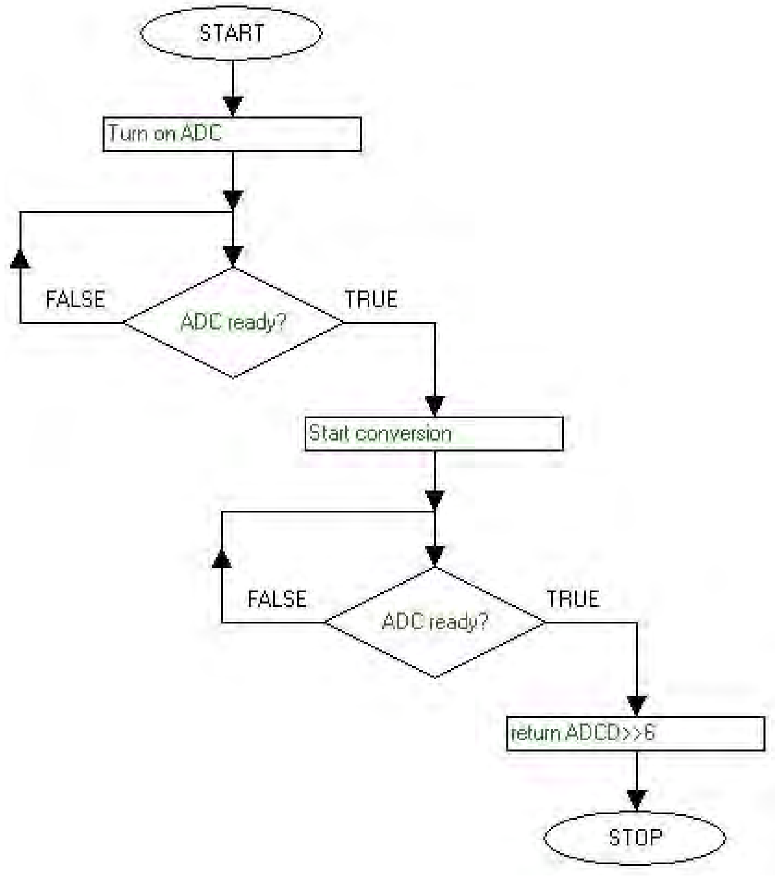


Figure 15. ReadPot() Function Flow

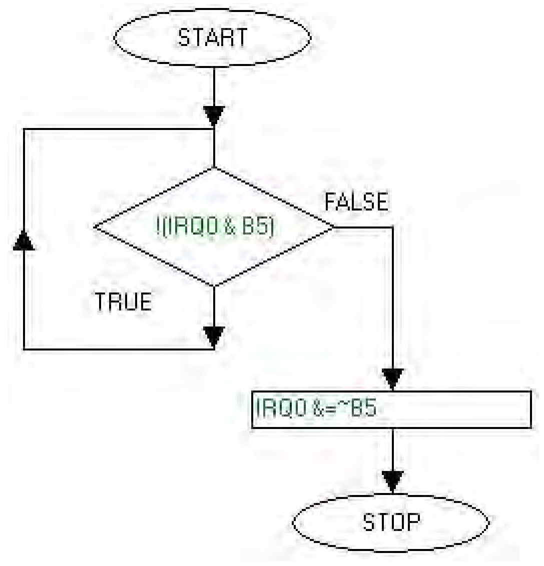


Figure 16. Wait() Function Flow

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.