Summary

This application note describes how to use the QED Board and the QED Digital I/O Board to directly control one or more stepper motors without the need for expensive indexers. The combination of a QED Board and one or two QED Digital I/O Boards delivers the combined capabilities of a controller, indexer and MOSFET driver for 1 to 5 motors at speeds to 1000 steps per second!

Description

A single QED Industrial Control System provides an enclosure for the QED boards, front panel keypad and display, serial connections for programming the system, and convenient screw terminals that make it easy to attach the opto-isolated 3 Amp drive signals to the stepper motors.

QED-3 Boards shipped with V3.11 or later of the kernel software contain the software support utilities as well as the new version of the onboard U1A PAL required to drive the stepper motors. QED-3 Boards shipped starting in June 1996 have these new features. Call Mosaic Industries if you are interested in upgrading previously purchased QED-3 Boards to run the stepper motor control code.

This document describes the stepper motor control software in detail, presents some sample commands, and includes a "mini-glossary" of pre-coded Forth and C motion control functions.

Overview of the QED Board's Stepper Motor Control Capabilities

Steppers are versatile motors that move in discrete steps and are well suited to digital motion control systems. The motors typically turn 1.8 degrees per step, or 200 steps per revolution, although different motors may have larger or smaller step sizes than this standard value.

Typical motor control systems involve:

- a computer/controller that issues high level commands to an indexer to manage the operation (such as moving an X/Y table, moving items from one location to another, controlling a cutting process, etc.);
- an indexer that issues step commands; and
- a driver that responds to the indexer by applying drive voltage and current to the motor coils.

These systems span a wide range of complexity and performance. At the high-performance end of the spectrum, "microstepping" drivers with high-voltage pulse-width-modulated drive can attain the ultimate in smoothness and quietness by driving the motor coils with analog sine waves as opposed to digital square waves. Simpler control schemes command the motor to move in full-step increments. "Halfstepping" control schemes allow the motor position to be incremented in half-step units which makes for greater smoothness and higher positioning precision compared to full-step control. Many motor control systems require a separate indexer and motor driver for each stepper motor in the system; consequently, the cost of multi-axis motion control systems can reach many thousands of dollars.

In most cases, stepper motors cannot instantly go from a stopped state directly to their maximum speed. Rather, they must "ramp up" in speed to attain the maximum speed. Similarly, a motor turning under load at full speed must be "ramped down" to a stopped state to maintain a correspondence between the physical position of the rotor and the number of applied steps.

The QED solution is ideal for non-microstepping applications that require 1 to 5 stepper motors at speeds up to 1000 full- or half-steps per second. Precoded software running on the QED Board performs the functions of the application software, high level motor control, and indexer. The indexer functions are interrupt-based, so other tasks in the application (data acquisition, calculation, display updating, etc.) can be performed while the motors are stepping. High level motion control functions make it easy to specify and control the speed and number of steps to be taken. Smooth ramp-up and ramp-down speed profiles are performed automatically based on acceleration and deceleration parameters set by the programmer for each motor.

The high current drivers on the QED Board and the QED Digital I/O Board perform the motor driver functions using MOSFETs to switch a DC voltage source through the motor coils. A QED Industrial Control System with a QED Board and two Digital I/O Boards can control and drive up to 5 stepper motors. While the 4 MOSFETs on the QED Board itself are limited to steady-state currents of 150 mA per coil, the 8 MOSFETs on each QED Digital I/O Board can control 3 Amps per coil at voltages up to 50 VDC. These 8 opto-isolated outputs on each QED Digital I/O Board can control two unipolar 4-coil stepper motors.

Overview of the Stepper Control Code

Three functions have been added to the QED V3.11 kernel to support stepper motor control. These QED-Forth functions are named CREATE.RAMP, SPEED.TO.DUTY, and STEP.MANAGER. The Control C versions of the functions are named CreateRamp(), SpeedToDuty() and StepManager(). A high level source code file (available in both Forth and C) provides a variety of motor control functions to manage ramping, speed and step control. Throughout this document both the Forth and C versions of the functions and high level commands will be presented.

The stepper code is explained in more detail in the Forth and C source code listings which contain the high level stepper control routines. These documents are available from Mosaic Industries. The availability of the high level source code allows you to customize the stepper control functions for your own application.

A brief overview of the high level source code is presented here along with the glossary entries for the three utility functions in the V3.11 PROM. Glossary entries for key high level stepper control functions are also included in this document.

Conceptually, the stepper motor control code comprises the following elements:

1. Two arrays of structures that are accessed by both the low-level support functions and the high level code. One array holds a status structure for each stepper motor, and the other array defines a speed ramp for each stepper motor.
2. Three support functions in the V3.11 PROM that perform the indexer functions and write the step patterns to specified high-current-driver output ports.
3. A high level source code file (available in Forth and C) that defines a set of useful motion control functions.

Stepper Motor Data Structures

A 1-dimensional array of status structures is called STATUS.ARRAY in Forth, and status_array in C. For each motor it specifies the extended address and bit mask of the port that controls the motor, the state of the motor (disabled, stopped, in a ramp, at a final speed, etc.), a signed 32 bit step counter, a direction flag, a set of bit masks that specify the step patterns, the default steady speed, the jog/start speed, the default acceleration and deceleration, additional parameters used by the utility routines, and a pointer to the RAMP.ARRAY.

The array that specifies the number of steps to be taken at each speed for each motor is called the RAMP.ARRAY (ramp_array in C). Each row corresponds to one stepper motor, and each entry in a row contains a two element structure that specifies a number of steps and a corresponding speed (represented as a duty cycle as explained below).

PROM-Resident Utility Functions

Two of the PROM-resident utility functions are used to write to the RAMP.ARRAY. SPEED.TO.DUTY (named SpeedToDuty() in C) converts a specified speed in steps per second to a "duty cycle" measure used internally by the motor control code. The

CREATE.RAMP function (named CreateRamp() in C) initializes the RAMP.ARRAY. It expects as input parameters the starting speed, ending speed, linear acceleration, ticks per second of the 68HC11's time base, a starting ramp address, and the number of entries in the ramp. It uses a linear ramp algorithm to initialize the appropriate elements in RAMP.ARRAY, and returns the total number of steps in the resulting ramp.

The third PROM-resident utility function is named STEP.MANAGER (StepManager() in C). This function is designed to be called from a periodic interrupt service routine; the default time base is once per millisecond. STEP.MANAGER expects the base address of the STATUS.ARRAY in the Y register. For each enabled motor, it writes the appropriate pattern at the appropriate duty cycle to the motor port to attain the speed specified in the motor's RAMP.ARRAY. This assembly coded routine executes in approximately 120 µs per enabled stepper motor. Thus running four stepper motors at a maximum speed of 1000 full- or half-steps per second requires approximately half of the 68HC11's available time (480 µs interrupt service time every 1000 µs).

High Level Stepper Motor Control Code

The high level control code declares and allocates the STATUS.ARRAY and RAMP.ARRAY, sets up the clock interrupt service routine using output compare 3 (this can be changed by the user), defines routines that initialize the STATUS.ARRAY, and defines a versatile set of functions to facilitate motion control. Sample function names are 1STEP, JOG.STEPS, STEPS.AT.SPEED, STEPS.AT.STEADY, CHANGE.SPEED, SOFT.STOP, and ESTOP (the corresponding names in Control C are similar). The well commented high level source code file and the descriptions below present more details regarding these functions.

How the Software Handles Stepping and Half Stepping

To "full step" a standard unipolar stepper motor, a sequence of 4 step patterns is applied in a specified order. The frequency at which the patterns are applied is the stepping frequency. In other words, one step is taken each time a new pattern is written to the motor port. Advancing through the 4 patterns in one direction produces clockwise motor rotation, and advancing through the patterns in the opposite direction produces counter-clockwise motor rotation.

To "half step" the same motor, a distinct sequence of 8 step patterns is applied in a specified order. Now each time a new pattern is written to the motor port, the motor moves through half the angle it would if it were full stepping.

You can configure any stepper motor for half stepping or full stepping when runtime initialization occurs. Half stepping may provide smoother performance in some applications, but for a given time base as set by the TICKS/SECOND (TICKS_PER_SECOND) constant, half stepping results in a slower attainable speed as expressed in revolutions per second for a given motor. This is because the low level utility functions do not distinguish between half steps and full steps. Thus, if the software is capable of writing a new step pattern to a motor 1000 times per second, then a motor configured for full stepping will take a maximum of 1000 steps per second, while a motor configured for half stepping will take a maximum of 1000 half steps per second. If these two motors are identical, the first will turn at twice the speed of the second.

All speeds, accelerations, decelerations and step counts are expressed in units of full steps or half steps, depending on how the stepper motor has been initialized. The InitSteppers() function at the bottom of the high level source code file corrects for this effect by applying a 2X correction factor to the speeds and accelerations saved in the status array if half stepping is specified. But the user must still be aware when specifying the number of steps or half steps, or when calling functions that take a speed as a parameter. The following table summarizes the units for some key parameters:

Parameter	Units	
	HalfStep Mode	FullStep Mode
Number of steps	Half steps	Steps
Speed	Halfsteps/sec	Steps/sec
Acceleration	Halfsteps/sec/sec	Steps/sec/sec
Deceleration	Halfsteps/sec/sec	Steps/sec/sec

A Tour of the High Level Stepper Motor Control File

The high level source code files named STEPPERS.4TH and STEPPERS.C provide a set of useful stepper control functions coded in Forth and C, respectively. You can customize the code for your own application by making some simple modifications to a few constants and functions.

Specifying the Number of Stepper Motors and the Speeds Per Ramp

Near the start of the file are some constant definitions. For maximum efficiency, set the constant MAX#STEPPERS (named MAX_STEPPERS in C) equal to the number of stepper motors that you are controlling. The source code file specifies 5 stepper motors: one is controlled by the 4 high current drivers on the QED Board, and each of two QED Digital I/O Boards controls two additional stepper motors.

There is a trade-off between the number of motors being controlled and the maximum stepping speed, limited by the available processing power of the 68HC11. Given that each interrupt-driven step requires 120 µsec per motor, while all 5 motors are stepping at their maximum speeds (1000 steps or halfsteps per second in this application), the 68HC11 processor is busy about 60% of the time managing the motion via the low-level interrupt-driven utilities. Thus most applications would either use fewer motors or would lower the TICKS/SECOND constant (TICKS_PER_SECOND in C) to decrease the maximum attainable speed of each motor. TICKS/SECOND is described in the "Clock Interrupt Service Routine" section below.

The constant SPEEDS.PER.RAMP (SPEEDS_PER_RAMP in C) specifies how many discrete speeds each ramp-up or ramp-down contains; the default value is 8. If you need smoother ramps containing more discrete speeds, you can make this value larger; values smaller than 8 are not recommended. For each incremental increase in this constant, the number of bytes in RAMP.ARRAY (ramp_array) is increased by MAX#STEPPERS * 2.

The Stepper Status Data Structure

The source code defines the STEPPER.STATUS (stepper_status in C) arrays of structures that hold all of the stepper motor control parameters that are described in the "Stepper Motor Data Structures" section above. The stepper status structure contains parameters that are used by the low level utility functions as well as useful high level information. The initialization functions that set up these parameters for each motor in use are located at the end of the source code file and are described later in this document. They are named INIT.STATUS and INIT.STEPPERS (InitStatus() and InitSteppers() in C).

Step Counter

One useful parameter in the stepper status structure is the signed 32-bit step counter that keeps track of the step count over a range of +/- 2,147,483,647 counts. Positive values represent clockwise rotation, and negative values represent counter-clockwise rotation. The following code fragments reference the 32-bit step count for motor 2.

In Forth:

```
2 STATUS.ARRAY +STEP.COUNTER 2@( -- d)
```

which leaves the value on the data stack.

In C:

```
Long current_step_counter =
      status_array[2].stepCounter;
```

which assigns the stepCounter to a variable.

You can set the step counter to any value you want as long as the motor is not moving. For example, if you perform a "HOME" operation in which a microswitch is used to detect the "home" position of stepper motor 2, you could then set the motor's step count to zero with the following code:

In Forth:

```
DIN 0 2 STATUS.ARRAY +STEP.COUNTER 2!
```

In C:

```
status_array[2].stepCounter = 0;
```

If the motor is configured for full stepping, the count indicates the number of full steps that have been taken since the status array was initialized or since the step counter was last zeroed (for example, during a "HOME" operation in which a microswitch is used to detect the stepper's "home" position, and the step count is set to zero when the switch is depressed). If the motor is configured for half stepping, the count indicates the number of halfsteps that have been taken. In each case the counter is a signed 32-bit value. To calculate the total number of revolutions, simply divide the step counter by the number of steps (or half steps) per revolution for your motor.

For example, if a particular motor is configured for full stepping and moves 1.8 degrees per step (a common value), then the motor has 200 steps per 360-degree revolution. Assume that we initialize the motor, set its step counter to zero at the motor's "home" location, and then command the motor to step 450 clockwise steps and 250 counter-clockwise steps. Reading the step counter using the code fragment above will return a value of +200 steps which is equivalent to exactly one clockwise revolution.

Number of Steps In Ramp Parameter

Another parameter maintained in the status structure array is an unsigned 16-bit step counter that is zeroed at the start of each ramp and incremented once for each step or half step taken. The high level application program may read but not write to this parameter, as it is used by the low level stepper motor control utilities. Its value for stepper motor number 3 can be obtained as follows:

In Forth:

```
3 STATUS.ARRAY +ELAPSED.STEPS.IN.RAMP @
      ( -- u)
```

which leaves the value on the data stack.

In C:

```
uint  current_steps_in_ramp =
    status_array[3].elapsedStepsInRamp;
```

which assigns the value to a variable.

The Jog/Start Speed

You may specify several default speeds and accelerations for each motor at initialization time. One user-specified speed parameter is called the jog/start speed; it should be chosen as a speed in the "safe starting" region of the motor's speed/torque characteristic. In other words, the motor should be able to abruptly move from a stopped state to a steady motion at the jog/start speed without losing any steps. The jog/start speed is used by the stepper control software for several purposes. First, when ramping up from a stopped state, the CREATE.RAMP function (which is called by all of the high level speed control functions) hops the motor speed right up to the jog/start speed, and then linearly increases speed (using the specified acceleration) to attain the specified final speed. Second, the jog/start speed is the default speed used by the JOG.STEPS function (named JogSteps() in C) to move from one position to another. Customize the INIT.STEPPERS (InitSteppers() in C) function at the end of the source code file to set the jog/start speed for each motor in your system.

The Steady Speed

Another user-specified speed parameter is called the steady speed; this is typically specified as the fastest speed that the motor can operate at smoothly under load without losing steps. The RAMP.TO.STEADY and STEPS.AT.STEADY functions (named RampToSteady() and StepsAtSteady() in C) use this speed as the final/maximum speed. Customize the INIT.STEPPERS (InitSteppers() in C) function at the end of the source code file to set the steady speed for each motor in your system.

The Acceleration and Deceleration

The other user-specified motion parameters are the acceleration (used during ramp-up to higher speeds) and the deceleration (used during ramp-down to lower speeds). These are used by the high level motion control functions to generate ramps during speed changes. For example, if a motor is full-stepping at 500 steps per second and its deceleration parameter is 2500 steps per second per second, a linear ramp-down to a stopped state will take 500/2500 = 0.2 seconds. A slower deceleration rate would result in a longer ramp-down period, and a faster deceleration rate would result in a shorter ramp-down period. Typically, lower acceleration/deceleration values are required for motors and loads with high inertia; the resulting slower ramp-up and ramp-down profiles avoid loss of steps during speed changes. Lightly loaded motors with low inertia can use higher accelerations and decelerations without losing steps. Customize the INIT.STEPPERS (InitSteppers() in C) function at the end of the source code file to set the acceleration and deceleration for each motor in your system.

Motor Port, Mask and Shadow RAM Parameters

The status structure array contains the 32-bit extended motor port address and a mask (typically the mask has 4 bits set and 4 bits clear) that specifies which bits of the port control the motor. These are set by the INIT.STATUS function (named InitStatus() in C). For efficiency reasons, the mask is actually the "non-motor mask": the 0s in the mask indicate the bits associated with the stepper motor, and the 1s indicate the non-motor bits.

Some output ports (such as those on the QED Digital I/O Board) are write-only ports. To keep track of the prior contents of the port, each port xaddress is associated with a "shadow RAM" address. This makes it possible to perform "read/modify/write" operations on the port that modify certain port bits while preserving the prior state of other bits.

The driver code for the Digital I/O Board defines shadow ram locations, and the high level stepper motor code also allows you to define shadow locations. Note that there must be only 1 shadow RAM byte per port! For example, if you are using the driver code for the QED Digital I/O Board as well as this stepper motor source code, you should use the appropriate reference to the OUTPUT.SHADOW (Output_Shadow in C) array as the shadow RAM addresses.

Some output ports (such as PPB on the QED Board) have built-in read-back capability. In this case the 16-bit shadow address must be set equal to the least significant 16 bits of the port xaddress. The source code file shows how to specify the correct port xaddress, non-motor mask and shadow address to interface a stepper motor to the QED Board's onboard high current drivers which are controlled by PPB.

Customize the INIT.STEPPERS (InitSteppers() in C) function at the end of the source code file to set the address, shadow, and mask information for each motor in your system. The default values presented at the end of the source code file provide a useful template.

Other Parameters in Stepper Status

The INIT.STATUS (InitStatus() in C) function described below initializes all of the elements in the stepper status structure array, and is described in more detail below. Other key parameters include the step patterns and number of step patterns, the specified direction (1 = clockwise, -1 = counter-clockwise), pointers to the ramp array, various low-level parameters, and the state variable. The state variable indicates whether the motor is disabled, stopped, at its final speed, in a ramp, about to enter a ramp, or about to exit a ramp. The source code defines named numeric constants to represent each state.

The Ramp Array

The RAMP.ARRAY (ramp_array in C) is a two-dimensional array of structures. Each row corresponds to one stepper motor. Each element in the row is a 4-byte structure comprising an integer "step limit" followed by an integer "target duty cycle". The step limit specifies how many steps are to be taken at each speed in the ramp, and the duty cycle is directly related to the speed. The CREATE.RAMP (CreateRamp() in C) function writes these parameters based on the specified start speed, steady speed and end speed, and acceleration for the motor. The low level STEP.MANAGER (StepManager() in C) function reads these ramp parameters to control the motor speed and step count.

The number of columns in the ramp array equals ((1+SPEEDS.PER.RAMP)*2); the SPEEDS.PER.RAMP constant is discussed in an earlier section. Each column can accommodate a ramp-up to a steady speed followed by a ramp-down to a terminal speed.

Clock Interrupt Service Routine

A periodic interrupt generated by an output compare (OC) is used to call the STEP.MANAGER function to control the motor speed. The frequency at which the service routine is called is set by the following related constants:

In Forth:
`TICKS/SECOND  TCNTS/TICK`

In C:
`TICKS_PER_SECOND  TCNTS_PER_TICK`

The default value of TICKS/SECOND is 1000; this limits the maximum motor speed to 1000 full- or half-steps per second. The related constant TCNTS/TICK specifies the number of counts of the processor's TCNT register per "tick" of the stepper interrupt. The TCNT register increments every 2 µsec, so for the default case there are 500 TCNTS per TICK. In general, the product of TICKS/SECOND times TCNTS/TICK should equal 500,000.

As described in the "Specifying the Number of Stepper Motors and the Speeds Per Ramp" section above, there is a trade-off between the number of motors being controlled and the maximum stepping speed, limited by the available processing power of the 68HC11. Each interrupt-driven step requires 120 µsec per motor, and it is generally a good idea to use less than 50% of the processor's time for a single interrupt function such as stepper motor speed control. Thus a typical application should use 4 or fewer stepper motors if TICKS/SECOND is set to 1000. When designing the application software, it is also important to make sure that competing interrupts do not disrupt each other. For example, interrupt-based control of several stepper motors might not be compatible with operation of the optional interrupt-driven secondary serial port at certain baud rates. The source code file includes a discussion of this point just before the definitions of the TICKS/SECOND constant.

The source code file uses OC3 (output compare 3) to generate the periodic stepper control interrupt. You may select a different output compare channel if you wish, but recall that OC2 is used for the multitasking executive's timeslicer, and OC4 is used to support the optional secondary serial port. The interrupt service routine simply loads the Y register with a pointer to the base of the ramp array, calls STEP.MANAGER (StepManager() in C), updates the TOC3 register, clears the interrupt flag bit, and returns. As described earlier, STEP.MANAGER performs the motor control functions based on the contents of the status array and the ramp array.

Step Pattern Sequences

The next section of the source code defines the step patterns for full stepping and half stepping. We assume that the stepper motor is interfaced either to the bottom 4 bits of a port (bits 0-3, called the "lower nibble") or the top 4 bits of a port (bits 4-7, called the "upper nibble"). Distinct constants are defined for the lower nibble and upper nibble patterns. The functions FULL.STEPS and HALF.STEPS (FullSteps() and HalfSteps() in C) are called by the INIT.STATUS (InitStatus()) function to initialize the step patterns in the stepper status array.

The step patterns required may vary according to the type of stepper motor used. You can edit the patterns according to the manufacturer's instructions for your stepper motor. For example, if a motor's sense of "clockwise" and "counter-clockwise" is wrong, you can reverse the order of the step patterns in the source code file to fix the problem.

Diagnostic Functions

The functions PRINT.1.RAMP and SHOW.STEPPER.STATUS (PrintRamp() and ShowStepperStatus() in C) are for diagnostic purposes only. Each function expects the motor index as the input parameter. The former function prints one row of the ramp array, and the latter function prints a summary of the stepper status structure for the specified motor. Calling these routines after executing various motor control functions may interest the curious user.

Speed Control Functions That Assume a Known Starting State

In this section we discuss some functions that control the motor speed assuming a known starting state (either stopped, or rotating at a known speed). Additional subsidiary functions are defined in the source code and may prove useful in some applications. The next section discusses higher level functions that are more flexible as well as functions that also control the number of steps taken.

Many of the motor control functions expect as input parameters a direction specifier and a motor index. The direction parameter is +1 to indicate clockwise rotation, and -1 to indicate counter-clockwise rotation. The CW and CCW constants are defined in the source code as +1 and -1, respectively. The motor index ranges from 0 to (MAX#STEPPERS - 1).

RAMP.TO.SPEED (RampToSpeed() in C) assumes that the motor is initially stopped. It expects the direction, target speed, and motor index as input parameters, and returns the number of full or half steps taken during the ramp-up. The motor is left running at the specified final speed. This function is used to ramp up from a stopped state to a final specified speed using the acceleration specified in the status array. As always, speeds are expressed in steps per second if the motor is configured for full stepping, and halfsteps per second if the motor is configured for half stepping.

For example, if motor 0 has been configured for full stepping using INIT.STEPPERS and is currently stopped, the following Forth command causes the motor to accelerate up to a speed of 420 steps per second in the counter-clockwise direction:

```
CCW  420  0  RAMP.TO.SPEED
```

The following statement has the same effect in C if InitSteppers() has been executed in main or elsewhere:

```
RampToSpeed( CCW, 420, 0);
```

(Note: if you are using Control C and are interactively calling a motor control function from the terminal, you must type the numeric direction indicators 1 or -1 instead of CW or CCW.)

RAMP.TO.STEADY (RampToSteady() in C) simply calls RAMP.TO.SPEED, specifying the motor's "steady speed" as defined in the status array as the target speed. (You specify the steady speed for each motor by customizing and executing INIT.STEPPERS.) RAMP.TO.STEADY expects the direction and motor index as input parameters, and returns the number of full or half steps taken during the ramp-up. This function is used to ramp up from a stopped state to the default steady speed.

FROM.SPEED.TO.STOP (FromSpeedToStop() in C) assumes that the motor is initially moving at a known speed. It expects the starting speed and motor index as input parameters, and returns the number of full or half steps taken during the ramp-down. The motor is left in a stopped, energized state with holding torque.

Highest Level Motion Control Functions

The section of the source code file titled "High Level Speed and Step Control" contains the most versatile motion control routines. Each of these functions is described in turn.

SOFT.STOP (SoftStop() in C) is the most versatile and safest way to stop without losing track of the step count. If the motor is already stopped or disabled, this function does nothing and returns a 0 to indicate that no steps were taken. If the motor is moving (either at steady speed or in a ramp), this routine

smoothly ramps it down to a stopped condition and returns the number of steps (if full stepping) or halfsteps (if halfstepping) that occurred during the ramp down.

ESTOP (EStop() in C) causes an abrupt "emergency stop" within 1 tick of the timebase interrupt, without a ramp down. It can be safely used to stop when stepping at jog speed, but at faster speeds, uncounted steps may occur due to motor inertia. ESTOP leaves the motor stopped with a step pattern applied so that the motor still has holding torque. It can also be used to undo the effect of disable_motor by energizing the motor coils (but note that before first using a motor, you must call INIT.STATUS and clear the motor port as done in INIT.STEPPERS).

CHANGE.SPEED (ChangeSpeed() in C) is the most versatile and safest way to attain any target speed regardless of the initial speed or state (i.e., STOPPED, IN_RAMP, etc.) of the motor. This routine can be used to ramp up to a specified speed from a stopped state, or ramp up or down to a new speed from a speed that is now in effect. It can even be called while the motor is in the middle of a ramp up or ramp down; this smart routine carefully avoids writing over a ramp that is currently in use by the background interrupt routine.

The remaining high level routines assume that the motor is stopped when the function is called:

- STEPS.AT.SPEED (StepsAtSpeed() in C) ramps up, steps at the user-specified speed, and ramps down for the total specified number of steps (if full stepping) or halfsteps (if halfstepping).
- STEPS.AT.STEADY (StepsAtSteady() in C) is very similar, except it uses the "steady speed" parameter that is stored in the motor's status structure as the stepping speed between the ramp up and ramp down.
- JOG.STEPS (JogSteps() in C) performs the specified number of steps at the jog/start_speed that is stored in the motor's status structure; no ramp-up or ramp down is performed because the jog speed is typically specified in the safe start/stop operating region of the motor.
- 1STEP (Step1() in C) simply performs 1 step in the specified direction.

Examples of High Level Motion Control Commands

For example, if motor 0 is configured for full stepping using INIT.STEPPERS, the following Forth command causes the motor to set up and execute a ramp to establish a speed of 420 steps per second in the current direction:

```
420  0  CHANGE.SPEED
```

The following statement has the same effect in C if InitSteppers() has been executed in main or elsewhere:

```
ChangeSpeed( 420, 0);
```

This command works whether the motor is stepping at a steady speed, ramping up, or ramping down. It even works if the motor is stopped (as long as it is not disabled and has taken at least one step since initialization); in this case the direction is the last direction that the motor was turning. The function returns the number of full- or half-steps that were performed during the transition ramp to the new speed.

To safely ramp down motor 3 to a stop from any speed, simply execute the Forth command:

```
3 SOFT.STOP
```

or the C command:

```
SoftStop(3);
```

The command uses the deceleration rate specified in the motor's status array, and the function returns the number of full- or half-steps that were performed during the ramp down.

As another example, if motor 2 is configured for half stepping, the following Forth command dictates a total of 10,000 halfsteps at a top speed of 900 halfsteps per second in the clockwise direction:

```
CW 10000 900  2  CHANGE.SPEED
```

The following statement has the same effect in C:

```
ChangeSpeed( CW, 10000, 900, 2);
```

Finally, if motor 1 is configured for full stepping, the following Forth command dictates 1,000 full steps in the clockwise direction at the jog/start speed specified in the motor's status structure:

```
CW 1000 1  JOG.STEPS
```

The following statement has the same effect in C:

```
JogSteps( CW, 1000, 1);
```

Recall that if you are using Control C and are interactively calling a motor control function from the terminal, you must type the numeric direction indicators 1 or -1 instead of CW or CCW.

Initialization Utilities

This final section of the source code allows you to define constants that specify your unique system configuration, including the port addresses that control the motors, the jog and steady speeds and acceleration parameters that are appropriate for each stepper motor. All initialization functions are performed by the function named INIT.STEPPERS (InitSteppers() in C). Let's examine some constant definitions and then step through the code for this initialization function.

System Configuration Port Addresses, Shadows, and Bit Masks

The system configured in the source code controls 5 stepper motors using a QED Board and 2 QED Digital I/O Boards. The 4 high current drivers on the QED Board control 1 motor, and each of the Digital I/O Boards' 8 high current driver ports controls 2 additional stepper motors. Thus the constant MAX#STEPPERS (MAX_STEPPERS in C) described above is set equal to 5. Many systems will involve fewer motors, and in these systems MAX#STEPPERS should be set to the appropriate value to avoid wasting time servicing motors that are not present.

One static variable is declared for each Digital I/O Board to serve as a "shadow" ram location for the write-only high current output port. Use of a shadow ram location allows the software to perform "read/modify/write" operations on the port; this means that we can modify some specified bits in the port while leaving others unchanged. The shadow variable for the first Digital I/O Board which controls motors number 0 and 1 is called MOTOR0&1.SHADOW in Forth, and motor0_1_shadow in C. The shadow variable for the second Digital I/O Board which controls motors number 2 and 3 is called MOTOR2&3.SHADOW in Forth, and motor2_3_shadow in C.

For proper operation, there must be one and only one shadow variable per output port. Note that the pre-coded driver software supplied by Mosaic Industries to run the Digital I/O Board defines shadow locations for each high-current output port, and the stepper motor source code file also defines shadow locations for each Digital I/O Board's high-current output port. If both of these source code files are loaded, at least one of them must be edited so that there is only one shadow variable per output port.

The stepper motor source code file next defines a series of constants that specify the extended address (address and page) and the bit mask that maps each stepper motor onto its associated output port. To maximize runtime efficiency, the bit mask is the "non-motor mask" with 0s indicating the motor bits and 1s indicating the non-motor bits. The QED Digital I/O Board's high current output port is accessed by writing to address 0003, and motors 0 and 1 are controlled by a Digital I/O Board mapped onto page DF (hex). Motor 0 occupies bits 0-3 and has a non-motor mask of hex F0, and motor 1 occupies bits 4-7 and has a non-motor mask of hex 0F. Motors 2 and 3 are controlled by another Digital I/O Board mapped onto page DE (hex).

Motor 5 is controlled by the 4 high-current drivers on the QED Board itself; these outputs are controlled by port PPB and the PAL, and are brought out to the keypad/display connector. The extended port address is hex 008081, and a separate shadow address is not needed, so the shadow address is also hex 8081. The correct non-motor mask for the QED Board's high current driver port is hex 1F. If these values for the port address, shadow, and non-motor mask are used, the driver software will correctly control the onboard high current drivers.

The appropriate constants for all of these parameters are defined in the source code file.

Specifying Default Speeds and Accelerations

As described above, each motor's stepper status structure includes a default jog/start speed, steady speed, acceleration, and deceleration. You can specify these parameters for each motor at the time of initialization.

The "jog/start speed" should be chosen as a speed within the "safe starting" region of the motor's speed/torque characteristic. This means that the motor must be able to directly transition from a stopped state to the jog/start speed without losing any steps. This speed is used by the JOG.STEPS (JogSteps() in C) function, and is also used as the starting speed in a ramp-up from a stopped state. Its units are steps per second if the motor is configured for full stepping, and halfsteps per second if the motor is configured for half stepping.

The "steady speed" is typically set equal to the maximum speed at which the motor can turn under load without losing any steps. It is used by the functions RAMP.TO.STEADY and STEPS.AT.STEADY (RampToSteady() and StepsAtSteady() in C). Its units are steps per second if the motor is configured for full stepping, and halfsteps per second if the motor is configured for half stepping.

The "acceleration" specifies the rate of change in speed used during a ramp up from a low speed to a higher speed. The "deceleration" specifies the rate of change in speed used during a ramp down from a high speed to a lower speed. The units are steps per second per second if the motor is configured for full stepping, and halfsteps per second per second if the motor is configured for half stepping.

The source code file defines named constants for each of these parameters that are used in the INIT.STEPPERS (InitSteppers() in C) function. In the source code example, motors 0, 1, 2, and 3 share the same parameters, while motor 4 has a different steady speed. Of course you can specify different parameters for each motor in your system.

The Initialization Function

The overall initialization function is named INIT.STEPPERS in Forth, and InitSteppers() in C. An examination of this routine illuminates all of the elements required for proper initialization of the stepper motor software.

In the source code example, the function expects a single flag that determines whether all of the motors are configured for full or half stepping. You may edit the routine to individually configure some motors for full stepping and others for half stepping. The local variable declaration at the top of the function defines a halfstep flag and a speedfactor which equals 1 for full stepping and 2 for half stepping. The speed factor converts all speeds used in the function to units of full steps per second, and all accelerations to full steps per second per second, whether or not the motor is configured for half stepping.

The function first calls DISABLE.STEPPER.IRQ (DisableStepperInterrupt() in C) to ensure that the stepper motor interrupt is not running while we write over all the control arrays and structures. Next it calls ZERO.RAMP&STATUS.ARRAYS (ZeroRampAndStatusArrays() in C) to start out with a known initialized condition in all of the arrays and structures.

The function then calls the INIT.STATUS (InitStatus() in C) routine five times, once for each motor in use. INIT.STATUS expects a set of input parameters and initializes the motor's stepper status structure accordingly. The input parameters are the jog/start speed, steady speed, acceleration, deceleration, port address and page, extended shadow address, non-motor mask, halfsteps flag, and the motor index. Note that all speeds and accelerations are multiplied by the speedfactor before being passed to INIT.STATUS.

After the final call to INIT.STATUS, INIT.STEPPERS (InitSteppers() in C) enters a loop that calls CLEAR.MOTOR.PORT (ClearMotorPort() in C) once for each motor. This is required to initialize the port and its shadow to a know state before attempting read/modify/write operations. If your system involves a motor that shares a port with active-low signals (that is, signals that must be initialized to the inactive high state), you can modify the source code to accomplish this. The important point is that the port and the shadow ram location must be written to ensure that they contain the same values at startup.

Only after all the motor ports have been initialized does the INIT.STEPPERS function enter a loop to call ENERGIZE.STEPPERS (EnergizeSteppers) for each motor. This applies a step pattern and confers holding torque on the motor. If you wish to initialize all of the stepper data structures but leave the motors un-energized, you can comment out this final loop. Be aware, however, that the first step command issued to an un-energized motor will simply energize the motor rather than cause a step; the second and subsequent step commands will result in motion.

The final command line in INIT.STEPPERS is a call to INIT.STEPPER.IRQ (InitStepperInterrupt() in C) which locally enables the stepper motor interrupt (OC3 in this example) and globally enables interrupts.

You're Ready to Go!

After interfacing your motors, customizing the system configuration and speed parameters for your system, and executing INIT.STEPPERS (InitSteppers() in C), you can try the motion commands listed above in the "Examples of High Level Motion Control Commands" section.

Stepper Motor Control Function Lists

The following lists summarize some of the key motion control functions available in the kernel and the high level source code, and present a very brief summary of the function's action. Not all functions and constants are summarized here. For more details consult the well documented source code file. Separate lists are presented for Forth and C.

In the Forth mini-glossary, the stack picture in parentheses describes the input and output parameters. All parameters are integers except those parameters that have the letters "addr" in their name, such as "xaddr" (an extended 32 bit address) or "start_ramp_addr" (a 16 bit address). The "#" symbol is pronounced "number".

Motor Control Functions: QED-Forth Mini-Glossary

+STEP.COUNTER (xaddr1 -- xaddr2)
32-bit signed integer (Long) number of steps in status array; can be read or written.

1STEP (direction\motor# --)
Assumes motor is stopped, takes 1 step.

CCW (-- -1)
A constant that indicates the counter-clockwise direction.

CHANGE.SPEED (target_speed\motor# -- #steps_in_ramp)
Most versatile speed control function, ramps up or down to the specified speed in the current direction.

CLEAR.MOTOR.PORT (motor# --)
Writes 00 to the output port and associated shadow ram byte that control the stepper motor.

CREATE.RAMP (start_speed\end_speed\accel\ticks/sec\start_ramp_addr\speeds/ramp -- #steps_in_ramp)
In PROM; writes speed parameters into motor's ramp array based on input parameters.

CW (-- +1)
A constant that indicates the clockwise direction.

DISABLE.MOTOR (motor# --)
Disables and de-energizes motor; calling this function boosts processor efficiency if the motor is not in use.

DISABLE.STEPPER.IRQ (--)
Locally disables the stepper service interrupt (OC3).

ENERGIZE.STEPPER (motor# --)
Writes step pattern to motor port, thereby applying holding torque.

ESTOP (motor# --)
"Emergency stop"; stops motor without ramp-down and leaves motor energized.

FROM.SPEED.TO.STOP (initial_speed\motor# -- #steps_in_ramp)
Ramps the motor from a constant speed down to a stopped (0 speed) state.

FULL.STEPS (non_motor_mask\motor# --)
Configures STATUS.ARRAY for full stepping based on user-specified step patterns.

HALF.STEPS (non_motor_mask\motor# --)
Configures STATUS.ARRAY for half stepping based on user-specified step patterns.

INIT.STATUS (jog_speed\steady_speed\accel\decel\port_addr\port_page\shadow_xaddr\nonmotor_mask\half\motor--)
Writes specified speed, acceleration, and step pattern information into STATUS.ARRAY.

INIT.STEPPER.IRQ (--)
Attaches and enables stepper service interrupt (OC3); globally enables interrupts.

INIT.STEPPERS (halfsteps? -- | note: stack picture depends on user customization)
Highest level initialization routine, should be customized by user for a given system.

JOG.STEPS (direction\#steps\motor# --)
Assumes motor is stopped, takes specified number of steps at jog/start speed.

MAX#STEPPERS (-- N)
A constant that specifies the number of stepper motors in use. Should be correctly set before compiling.

RAMP.ARRAY (motor#\ramp_index -- xaddr)
2-dimensional array of RAMP.ELEMENT structures, set by CREATE.RAMP, used by STEP.MANAGER.

RAMP.TO.SPEED (direction\target_speed\motor# -- #steps_in_ramp)
Ramps the motor from a stopped state to a specified speed.

RAMP.TO.STEADY (direction\motor# -- #steps_in_ramp)
Ramps the motor from a stopped state to the steady speed stored in STATUS.ARRAY.

SET.DIRECTION (direction\motor# --)
Writes to +DIRECTION element in STATUS.ARRAY. CW = +1; CCW = -1.

SOFT.STOP (motor# -- #steps_in_ramp)
Most versatile stopping function, ramps the motor down to a stopped (0 speed) state.

SPEED.TO.DUTY (steps_per_sec\ticks_per_sec -- duty_cycle)
In PROM; converts speed to the duty cycle representation used by STEP.MANAGER.

START.RAMP (direction\motor# --)
Called by higher level functions to start a ramp that has already been set up.
STATUS.ARRAY (motor# -- xaddr)
1-dimensional array of STEPPER.STATUS structures, initialized by INIT.STEPPERS.
STEP.MANAGER (--)
In PROM; core of interrupt service routine to service steppers; worst case execution time is appx. 120µs/motor.
STEPS.AT.SPEED (direction\#steps\speed\motor# --)
Assumes motor is stopped, ramps to specified speed and back to zero to achieve specified number of total steps.
STEPS.AT.STEADY (direction\#steps\motor# --)
Assumes motor is stopped, ramps to "steady" speed and back to zero to achieve specified number of total steps.
TICKS/SECOND (-- N)
Constant that sets stepper timebase and sets maximum number of steps or halfsteps/sec.

Motor Control Functions: Control C Mini-Glossary

int **CCW**
A constant equal to -1 that indicates the counter-clockwise direction.

uint **ChangeSpeed**(uint target_speed, int motor)
Most versatile speed function, ramps up or down to the specified speed in the current direction; returns #steps.

void **ClearMotorPort**(int motor)
Writes 00 to the output port and associated shadow ram byte that control the stepper motor.

uint **CreateRamp**(uint start_speed,uint end_speed,uint accel,uint ticks_per_sec, RAMP_ELEMENT* start_ramp_addr, uint speeds_per_ramp)
In PROM; writes speed parameters into motor's ramp array based on input parameters.

int **CW**
A constant equal to +1 that indicates the clockwise direction.

void **DisableMotor**(int motor)
Disables and de-energizes motor; calling this function boosts processor efficiency if the motor is not in use.

void **DisableStepperInterrupt**(void)
Locally disables the stepper service interrupt (OC3).

void **EnergizeStepper**(int motor)
Writes step pattern to motor port, thereby applying holding torque.

void **EStop**(int motor)
"Emergency stop"; stops motor without ramp-down and leaves motor energized.

uint **FromSpeedToStop**(uint starting_speed, int motor)
Ramps the motor from a constant speed down to a stopped (0 speed) state; returns number of steps in ramp.

void **FullSteps**(uchar non_motor_mask,int motor)
Configures STATUS.ARRAY for full stepping based on user-specified step patterns.

void **HalfSteps**(uchar non_motor_mask,int motor)
Configures STATUS.ARRAY for half stepping based on user-specified step patterns.

void **InitStatus**(uint jog_speed,uint steady_speed,uint accel,uint decel,xaddr port_xaddr,char* shadow,\ uchar nonmotor_mask,int half_steps,int motor)
Writes specified speed, acceleration, and step pattern information into STATUS.ARRAY.

void **InitStepperInterrupt**(void)
Attaches and enables stepper service interrupt (OC3); globally enables interrupts.

void **InitSteppers**(int halfstep) // note: stack picture depends on user customization
Highest level initialization routine, should be customized by user for a given system.

void **JogSteps**(int direction, uint numsteps, int motor)
Assumes motor is stopped, takes specified number of steps at jog/start speed.

int **MAX_STEPPERS**
A constant that specifies the number of stepper motors in use. Should be correctly set before compiling.

uint **RampToSpeed**(int direction, uint target_speed, int motor)
Ramps the motor from a stopped state to a specified speed, and returns number of steps in ramp.

uint **RampToSteady**(int direction, int motor)
Ramps the motor from a stopped state to the steady speed stored in STATUS.ARRAY; returns number of steps.

RAMP_ELEMENT **ramp_array**[][]
2-dimensional array of RAMP.ELEMENT structures, set by CREATE.RAMP, used by STEP.MANAGER.

void **SetDirection**(int direction,int motor)
Writes to +DIRECTION element in STATUS.ARRAY. CW = +1; CCW = -1.

uint **SoftStop**(int motor)
Most versatile stopping function, ramps the motor down to a stopped state; returns number of steps in ramp.

uint **SpeedToDuty**(uint steps_per_sec, uint ticks_per_sec)
In PROM; converts speed to the duty cycle representation used by STEP.MANAGER.

void **StartRamp**(int direction, int motor)
Called by higher level functions to start a ramp that has already been set up.

struct stepperStatus **status_array**[MAX_STEPPERS]
1-dimensional array of STEPPER.STATUS structures, initialized by INIT.STEPPERS.

void **Step1**(int direction, int motor)
Assumes motor is stopped, takes 1 step.

long **stepCounter**
32-bit signed integer (Long) number of steps in step_status struct in status_array; can be read or written.

void **StepManager** (void)
In PROM; core of interrupt service routine to service steppers; worst case execution time is appx. 120µs/motor.
void **StepsAtSpeed**(int direction, uint numsteps,uint target_speed, int motor)
Assumes motor is stopped, ramps to specified speed and back to zero to achieve specified number of total steps.
void **StepsAtSteady**(int direction, uint numsteps, int motor)
Assumes motor is stopped, ramps to "steady" speed and back to zero to achieve specified number of total steps.
int **TICKS_PER_SECOND**
Constant that sets stepper timebase and sets maximum number of