TI Designs

High Availability Industrial High Speed Counter (HSC) / Pulse Train Output (PTO)

Design Overview

This TI design provides a reference solution (firmware and test platform) for two different industrial IO functions related to motion control: High Speed Counter (HSC) and Pulse Train Output (PTO). The design is based on a microcontroller platform that is suitable for use in industrial applications where high availability and/or functional safety are also important requirements.

Design Resources

TIDM-HAHSCPTO	Tool Folder Containing Design Files
LAUNCHXL2-RM57L	Tool Folder
RM57L843	Product Folder
DP83630	Product Folder
INA210	Product Folder
LM26420	Product Folder
LM4040D20	Product Folder
TM4C129ENCPDT	Product Folder
TPD4E004	Product Folder
TPS2553	Product Folder
TPS3106K33	Product Folder

TI E2E™ Community

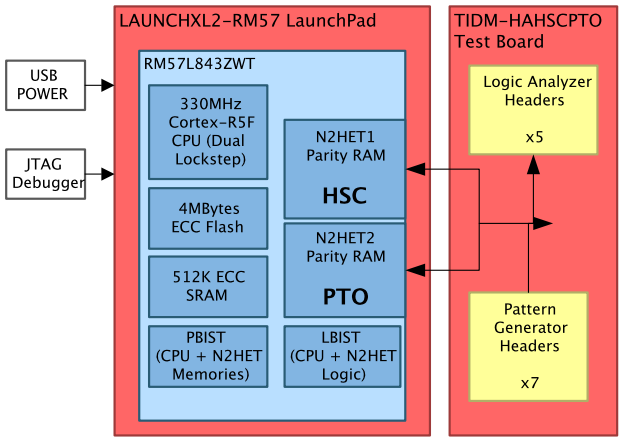
ASK Our E2E Experts
WEBENCH® Calculator Tools

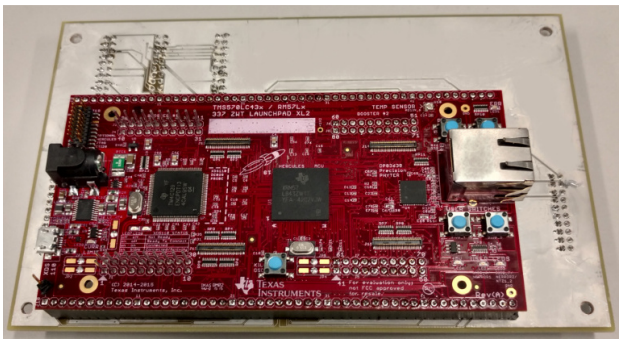
Design Features

- High speed (up to 400 KHz) 32-bit HSC counter with two counter input pins, four auxiliary input pins, two input capture registers, and two compare output pins.
- PTO generates output pulse trains and trapezoidal profiles up to 100 KHz with 9.1 ns resolution.
- Count/Dir, Clockwise/Counter Clockwise, and Quadrature Modes.
- Reduces CPU load and interrupt latency requirements by operating autonomously on the N2HET Timing Coprocessor.
- Reduces system cost by replacing FPGA or ASIC.
- Includes N2HET firmware and test platform, as well as host CPU driver functions.

Featured Applications

- Factory Automation and Process Control
- Programmable Logic Controllers (PLC)
- Motor and Stepper Drives
- Motion control applications requiring high availability or functional safety.





An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

1 Key System Specifications

Table 1. High Speed Counter (HSC) Specifications

PARAMETER		SPECIFICATION	DETAILS
Number of Counting Channels		2	
Physical Inputs Per Channel		6	2 Counting Plus 4 Auxiliary Inputs
Physical Outputs Per Channel		2	
Input Voltage Range		0 to 3.3 V[(1)]	
High Level Input Voltage (Min)		2.0 V[(1)]	
Low Level Input Voltage (Max)		0.8 V[(1)]	
High Level Output Voltage (Min)		3.0 V @ 50 µA[(1)]	
Low Level Output Voltage (Max)		0.2 V @ 50 µA[(1)]	
Input Frequency (Max)		400KHz	
Input Filtering	Inputs INA, INB	0 - 9.3 µs	HW Filtering - 10 bit counter
	Inputs INW, INX, INY, INZ	0, 1.16 us, 2.32 µs, ...	Programmable filter length to > 30 seconds in 1.16 µs increments.
Counter Modes		Count / Direction x1, x2	
		Clockwise / Counter Clockwise x1, x2	
		Quadrature x1, x2	
Counter Range		0 to 2^{32}-1 Counts	Counter Min, Max are programmable.
Counter Behavior at Min, Max Limits		Rollover or Saturation	Counter marked invalid on saturation.
Counter Hysteresis (On Direction Change)		0 to 2^{25}- 1 Counts	Programmable for each direction change.
Counter Reset		Optional, Programmable 32-Bit Reset Value	Programmable Triggering For each Function based on Software or Auxiliary Input (Edge/Level) Events.
Counter Sync		Optional, Programmable 32-Bit Sync Value	
Counter Enable		Optional	
Capture Registers		2x 32-Bit	
Output Compare Registers		4x 32-Bit	2 Compare Registers / Output Pin
N2HET Resource Usage	Memory	< 128 Words	Version 1.0.0: 107
	Execution Cycles	< 128 Cycles/Loop (max)	Version 1.0.0: max 119 cycles/loop

(1) Parameter directly from RM57L843 datasheet. In case of conflicts the datasheet value supersedes the values listed here.

Table 2. Pulse Train Output (PTO) Specifications

PARAMETER		SPECIFICATION	DETAILS
Number of Physical Outputs		2	
High Level Output Voltage (Min)		3.0 V @ 50 µA[(1)]	
Low Level Output Voltage (Max)		0.2 V @ 50 µA[(1)]	
Output Frequency (Max)		100 KHz	(2 PTOs /N2HET. 200 KHz if 1 PTO/N2HET)
Output Modes		Count / Direction	
		Clockwise / Counter Clockwise	
		Quadrature	
Output Step Direction		Forward, Reverse, Delay	Delay: Time Delay without Stepping
Output Acceleration		Linear Accelerate, Linear Decelerate, Constant Speed	
Steps Per Command		0 - 1M	
N2HET Resource Usage	Memory	< 80 Words	v1.0.0: 79 Words
	Execution Cycles	< 64 Cycles / Loop	v1.0.0: max 57 cycles/loop

(1) Parameter directly from RM57L843 datasheet. In case of conflicts the datasheet value supersedes the values listed here.

2 System Description

The High Speed Counter and Pulse Train Output modules can be used together as the basis of a motion control system. Figure 1 shows an example of such a system. The pulse train output module is used to generate a motion profile that drives the motor and conveyer. The actual speed and position of the system is sensed by an incremental encoder and tracked by the high speed counter through its main counter inputs INA, INB. The auxiliary inputs of the HSC can be used for capturing the encoder position output when a discrete input switch is closed as shown in this example, or an auxiliary input could be to used perform a homing function where the homing signal comes from a switched input. The HSC also includes two output compare pins that can be used to trigger turn on and off an actuator within a range of count values.

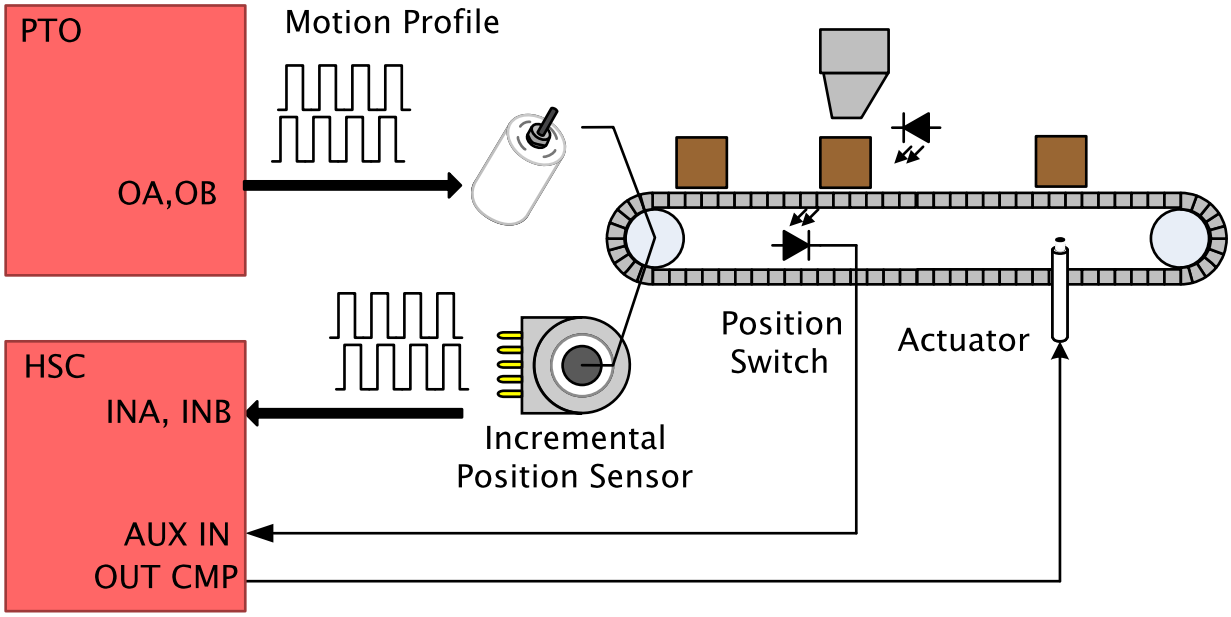


Figure 1. Example Motion Control System Using HSC & PTO Modules

2.1 RM57L843ZWTT with Dual N2HET Timing Coprocessors

The RM57L843 device (Figure 2) is part of the Hercules™ RM series of high-performance ARM® Cortex®-R-based MCUs. This product line is supported with SafeTI™ Design Packages for Functional Safety Applications. Comprehensive documentation, tools, and software are available to assist in the development of IEC 61508 functional safety applications, including:

- Safety Manual
- Detailed Safety Analysis Report with FMEDA (request form)
- Safety Analysis Report Summary (request form)
- Hercules SafeTI Diagnostic Library
- SafeTI Compliance Support Package for Hercules Diagnostic Library
- SafeTI Compliance Support Package for HALCoGen (Hardware Abstraction Layer Code Generator)
- SafeTI™ Compiler Qualification Kit for Hercules™ MCUs
- Standards Compliance for SafeTI Products and/or Processes (1)

The RM57L843 device has on-chip diagnostic features including: dual CPUs in lockstep, Built-In Self-Test (BIST) logic for CPU, the N2HET coprocessors, the for on-chip SRAMs. The L1 caches, L2 Flash, and L2 SRAM memories are ECC protected. The device also supports ECC or parity protection on peripheral memories and loopback capability on peripheral I/Os.

(1) IEC61508 SIL3 certificate for the RM57L843 device is expected before the end of 2016, certificates for other devices in the Hercules family are available today.

The N2HET is an advanced programmable timer. With the N2HET timer, the application can perform sophisticated timing functions with minimal CPU interaction or loading. Each N2HET on the device includes its own specialized processing unit, program/data memory, and a dedicated IO port that allows one timer to control up to 32 IO pins. The N2HET instruction set includes instructions for basic timing operations (counters, capture registers, compare registers) as well as arithmetic, logical, shift, and branch capabilities. Sophisticated timing functions, such as the HSC and PTO applications of this design, can be created from these instruction primitives. At the same time, these functions operate largely autonomously which provides two major benefits to the application developer when compared against a solution consisting of a CPU and a fixed function timer. First, the CPU loading for timing functions as a percent of the available CPU bandwidth is reduced, as fewer interactions are required. Second, there is often the possibility to relax the interrupt latency requirement that the timing function places on the CPU because the N2HET can execute a series of simple steps autonomously.

The N2HET also includes a transfer unit (the HTU) that allows it to initiate DMA operations to/from the main memory on the device. With the HTU, the N2HET can execute entire lists of commands from main memory or store a large number of samples to memory without any CPU interaction.

The high level of diagnostic coverage and error correction available on the RM57L843 device make the microcontroller suitable for high performance real-time control applications with functional safety and/or high availability requirements.

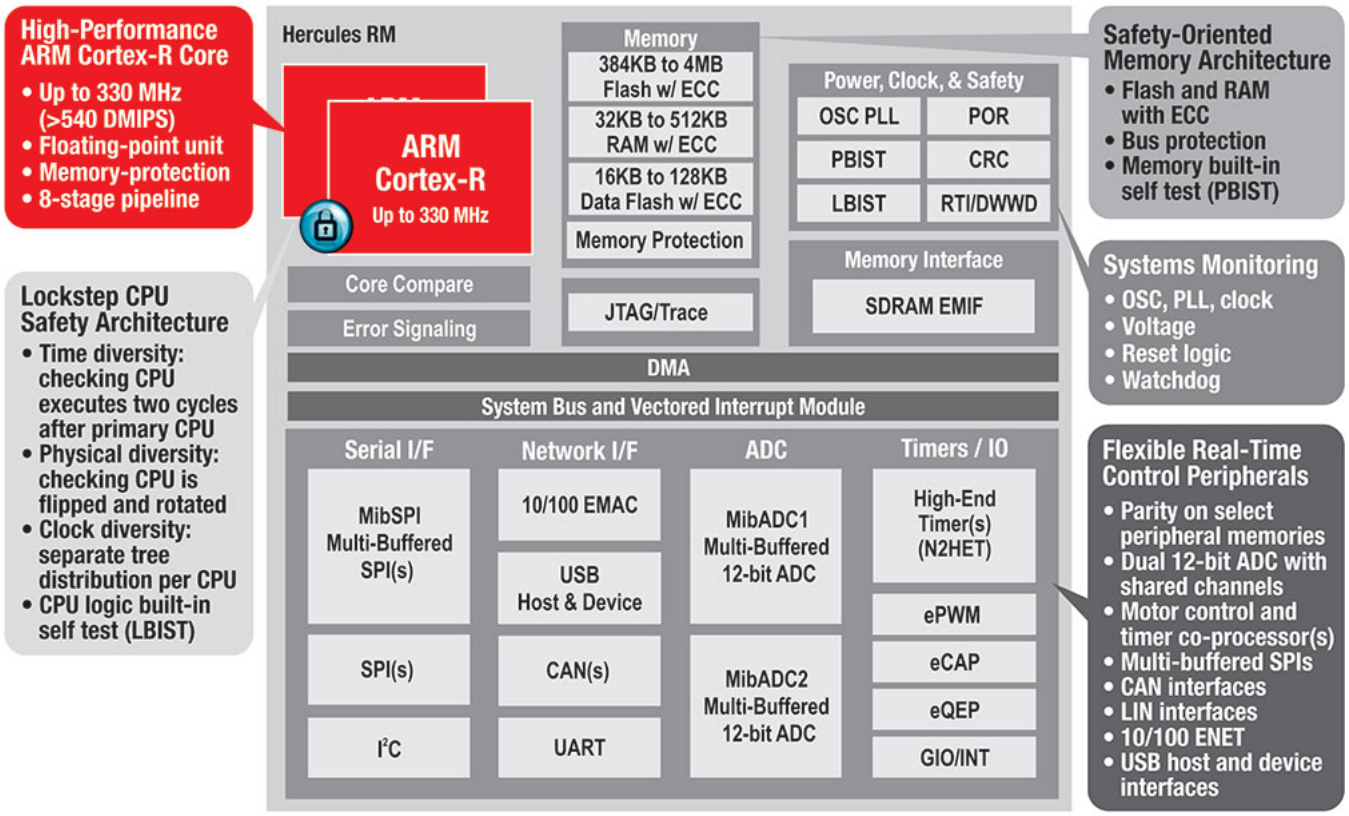


Figure 2. RM57L843 Block Diagram

2.2 LAUNCHXL2-RM57L: RM57L843ZWTT LaunchPad™

The Hercules RM57Lx LaunchPad Development Kit is based on the highest performance Hercules MCU RM57L843 – lockstep cached 330MHz ARM Cortex-R5F based RM series MCU.

The LaunchPad features connectivity options such as IEEE 1588 precision time Ethernet PHY DP83630 and has the capability in addition to the standard BoosterPack headers, for further expansion to an FPGA or an external SRAM using high density connectors for MCU's parallel interfaces - EMIF, RTP and DMM.

The RM57Lx MCUs include many diagnostic features such as ECC protection for CPU caches & other memories and a rich set of peripherals such as two 12-bit ADCs, programmable High-End timers, motor control peripherals (eQEP, eCAP, ePWM), Ethernet, MibSPI, EMIF and many serial communication interfaces.

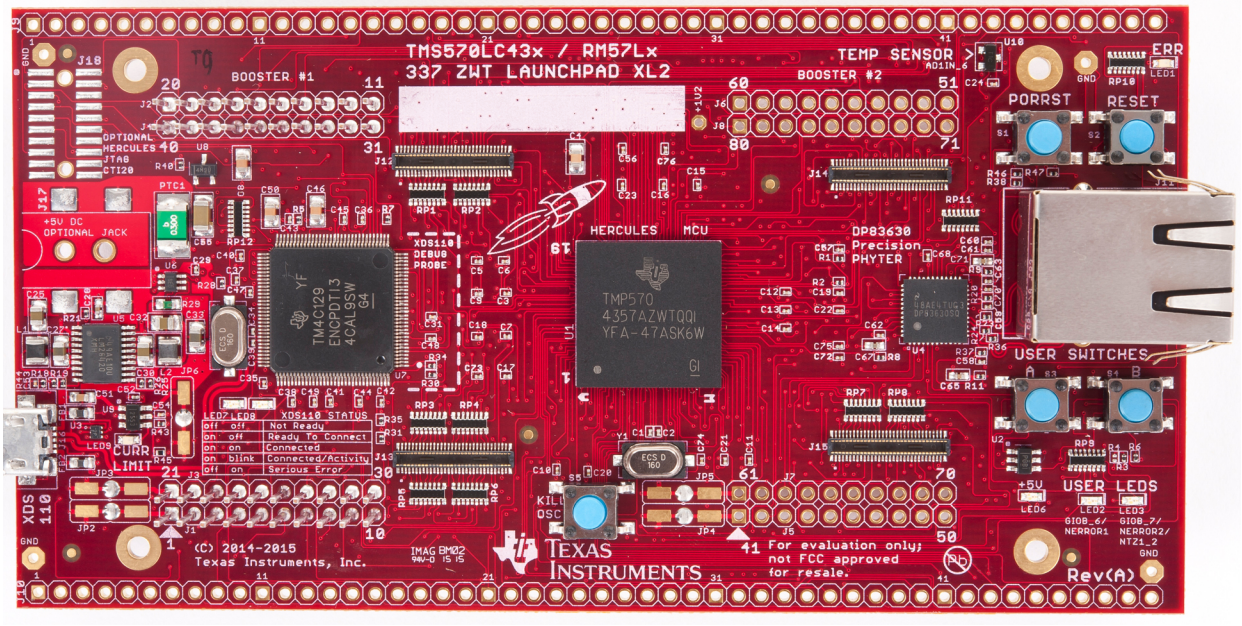


Figure 3. LAUNCHXL2-RM57L LaunchPad Featuring the RM57L843

3 Block Diagram

This section explains the block diagrams of the N2HET based HSC and PTO functions of this design.

3.1 N2HET Based High Speed Counter

Figure 4 contains a block diagram of the HSC function. It consists of four major sub-blocks:

- Counter Block
- Auxiliary Input Block
- Output Compare Block
- Input Capture Block

While this example maps the HSC function to N2HET1 on the RM57L843, the same function could also be mapped to N2HET2 or to both N2HET1 and N2HET2 if two HSC instances are required. The HSC timing specifications provided in Table 1 are based on the 110MHz maximum N2HET clock frequency available on the RM57L843; for other devices in the Hercules family the timing specifications can be scaled to match the N2HET frequency of a particular device. The N2HET memory and cycle requirements for each HSC instance are also listed in Table 1; so that it is possible to evaluate the number of HSC instances that can be implemented on a single N2HET. The HSC implementation does require more than 100 of the maximum 128 execution cycles available during each loop resolution period when the N2HET HR prescale is set to divide by 1 for maximum resolution. On the RM57L843 there is sufficient program memory on each N2HET to implement two instances of HSC on each N2HET, but the HR prescale must be increased to divide by 2 to allow for up to 256 execution cycles per loop resolution period. If this is done, then most [2] of the timing performance specifications of the HSC as described in Table 1 will be reduced by a factor of two.

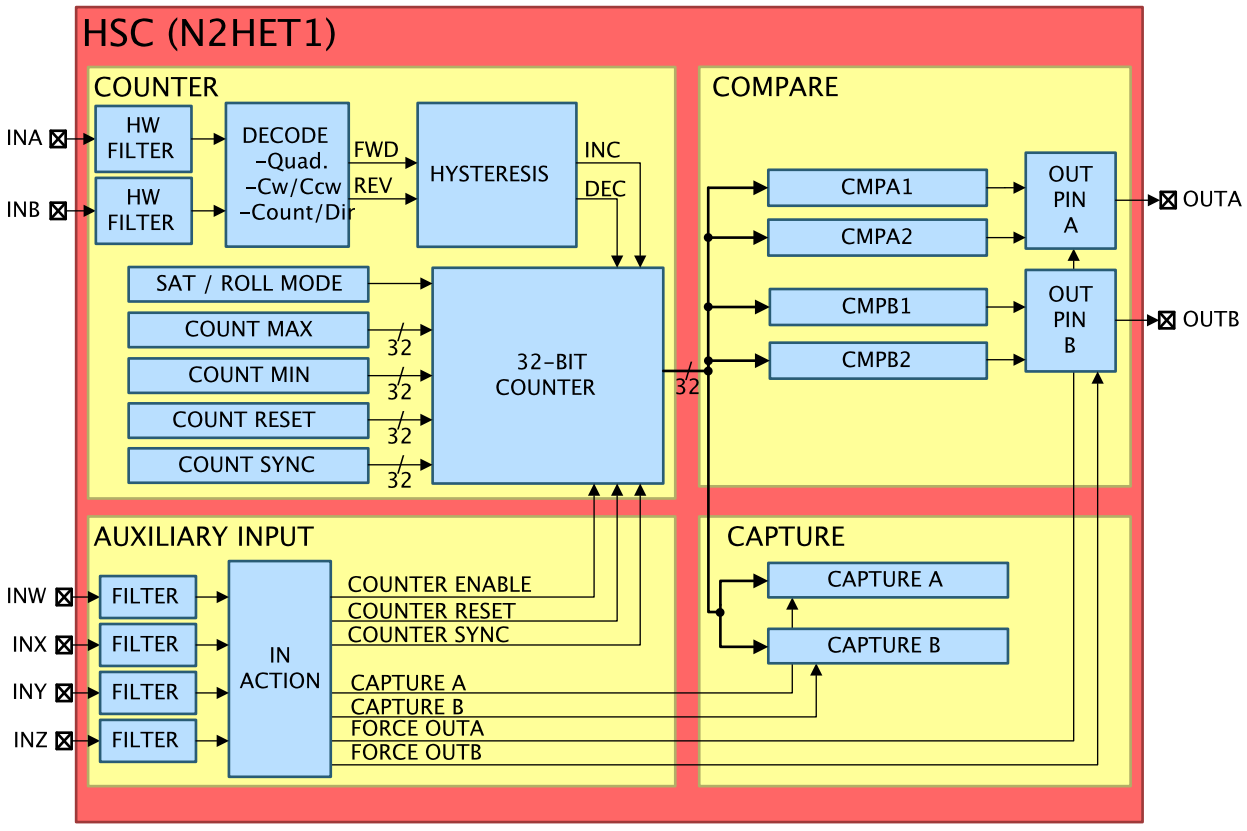


Figure 4. HSC Block Diagram

[2] An exception to this rule is the specification for hardware debounce, which is based on the VCLK frequency not the N2HET HR clock frequency.

3.1.1 HSC Counter Block

The counter block processes the two primary HSC inputs, INA and INB, and outputs a 32-bit count value that tracks the activity on the primary inputs. The counter block also takes auxiliary inputs from the auxiliary input block that control whether the counter is enabled, and whether or not it should be reset or loaded with a sync (homing) value.

The counter range may be programmed through two registers, MIN and MAX. The behavior of the counter when it reaches these limits is programmable. The counter may either saturate or rollover.

When the counter is programmed to rollover, an increment command from MAX results in a value of MIN. Likewise a decrement from MIN results in MAX.

When programmed to saturate, an increment from MAX results in the counter saturating at MAX. Likewise a decrement from MIN results in the counter saturating at MIN. When the counter saturates it also sets its internal valid state to indicate that the count is invalid.

When the counter is invalid, the output compare block actions are skipped. The counter may be returned to valid either by a synchronization event (homing) or through software running on the host CPU.

The primary counter inputs INA and INB use the N2HET hardware input filtering to debounce these pins. The debounced signal is decoded based on the counting mode. Three major counting modes are supported, where the major mode is determined by the counting waveform encoding. Each major mode supports two or three minor modes, which determine which edges of the waveform are counted:

- Count/Dir Major Mode (See Figure 5). x2 and x1 Minor Modes Are Supported.
- Clockwise/Counter Clockwise Major Mode (See Figure 6) x2 and x1 Minor Modes Are Supported.
- Quadrature Major Mode (See Figure 7) x4, x2 and x1 Minor Modes Are Supported.

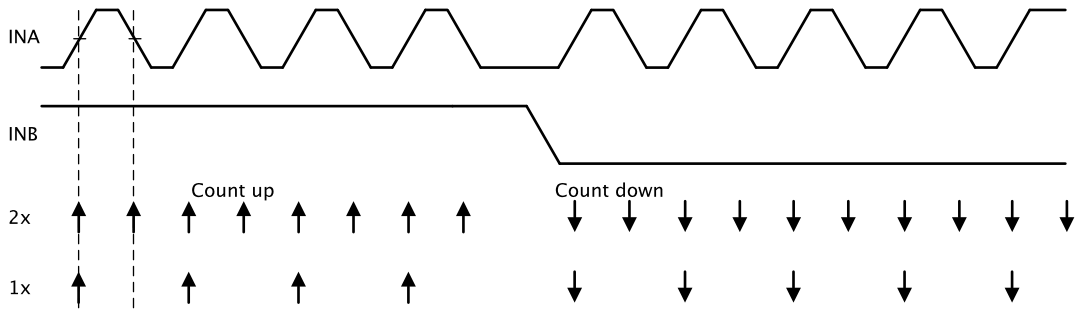


Figure 5. Counting in Count/Dir Mode Major Mode -x2, x1 Minor Modes

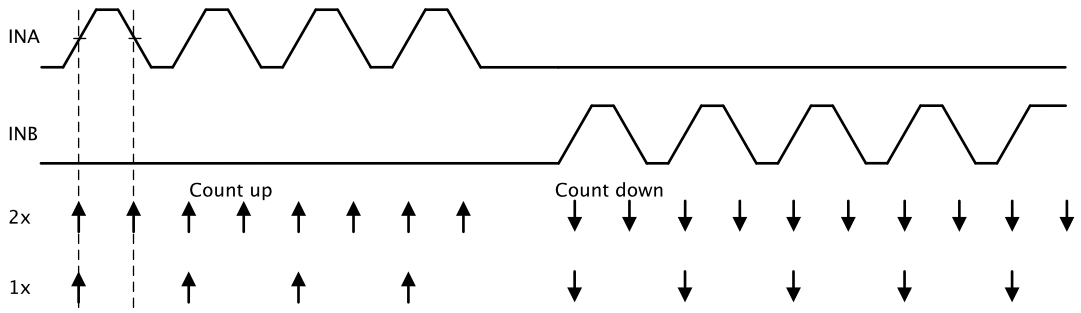


Figure 6. Counting in Clockwise / Counter-Clockwise Major Mode - x2, x1 Minor Modes

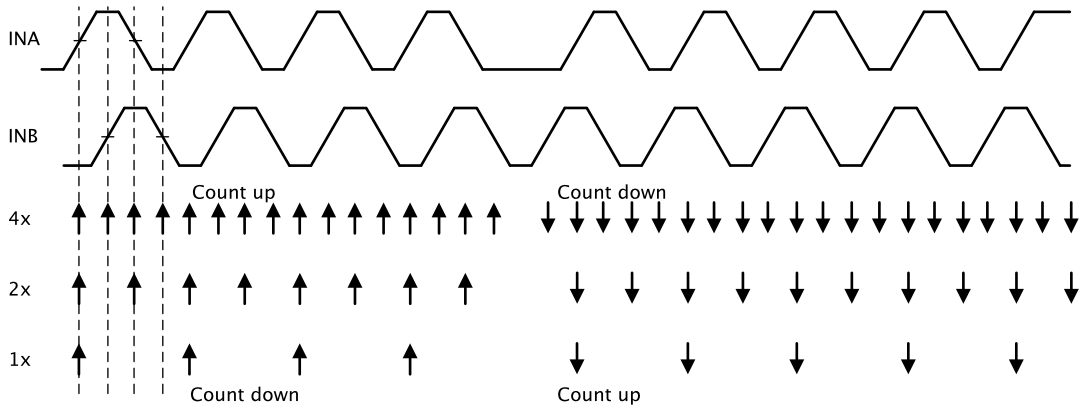


Figure 7. Counting in Quadrature Major Mode - x4, x2, x1 Minor Modes

After filtering and decoding the INA, INB primary inputs, the count up/down signal from the decoder is passed through a hysteresis block before the final increment or decrement command is issued to the counter. Hysteresis allows the counter to correct automatically for errors due to torsion or backlash in the mechanical driveline whenever the direction of motion is changed. The number of counts to filter on a direction change is programmable and can be different for each direction of change. Figure 8 shows an example where 3 counts of hysteresis is applied upon a direction change from incrementing to decrementing in quadrature x4 counting mode.

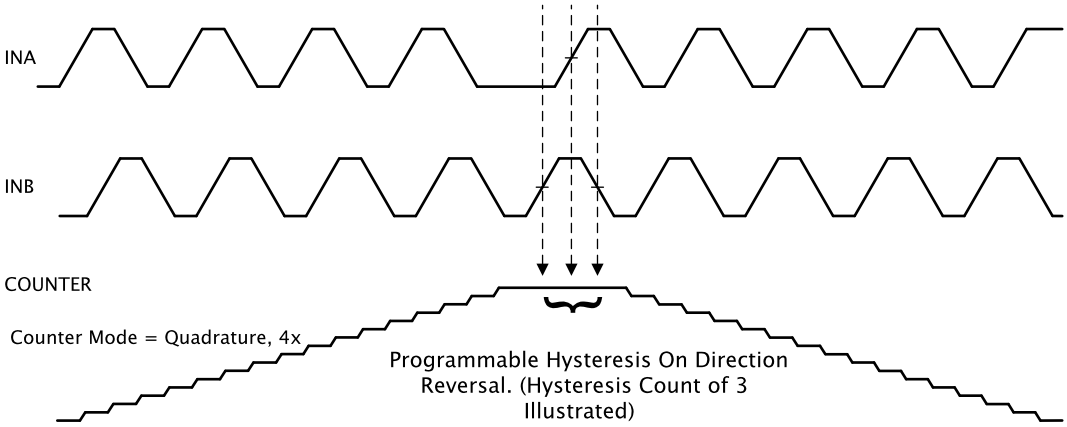


Figure 8. Example of Hysteresis on Direction Reversal

3.1.2 HSC Auxiliary Input Block

The auxiliary input block processes four auxiliary inputs (INW, INX, INY, and INZ) and performs the following functions:

- Samples each auxiliary input at the rate of the N2HET loop resolution (1.16 µs for this design).
- Filters each pin with a programmable filter delay.
- Delays the HSC startup until each auxiliary input is debounced at least once.
- Detects the following conditions for each pin: Rising Edge, Falling Edge, High Level, Low Level.
- Outputs pin action signals to the Counter, Output Compare, and Input Capture Blocks.
- Accepts software triggers for the pin action signals.

The signals provided to the Counter block are:

- Counter Enable - Must be active for the counter to count.
- Counter Reset - Causes the counter to reset to a predefined value (normally zero).

- Counter Sync - Causes the counter to preset to a predefined value.

The signals provided to the Capture block are:

- Trigger Capture A to Capture Counter Value
- Trigger Capture B to Capture Counter Value

The signals provided to the Output Compare block are:

- Force OUTA to a predefined state (High or Low).
- Force OUTB to a predefined state (High or Low).

Each of these signals is formed by a logical combination of the INW, INX, INY, and INZ pin conditions (after any filtering has been performed). Additionally there are software triggered conditions for the counter enable, reset, sync, input captures, and output force functions. The software counter enable and output force bits are persistent - they remain in effect until cleared by software. The software reset, sync, and capture conditions are one-shot, they are set by software and automatically cleared after being processed.

The Auxiliary input block combines the pin conditions and software conditions with a bitwise OR-AND structure. First, two sets of OR conditions across all pin and software conditions are processed to produce two pattern matches. The logical AND of each pattern match then produces the final trigger event out of the Auxiliary input block. Figure 9 illustrates the OR-AND structure that can be configured for each of outputs of this block.

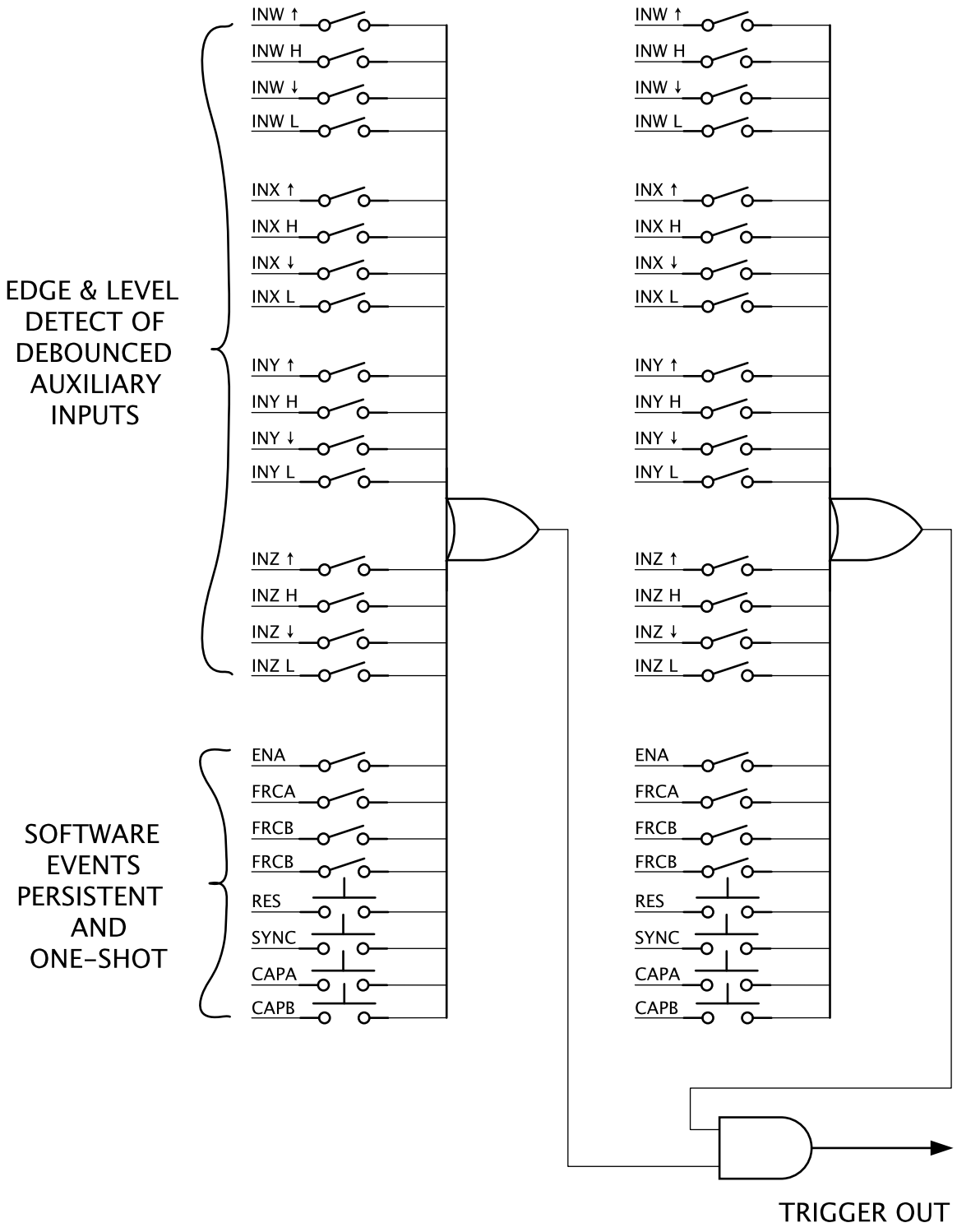


Figure 9. Trigger Options based on Auxiliary Inputs and Software Events

3.1.3 HSC Output Compare Block

The HSC Output compare block implements two output compare registers for each HSC output pin. The comparisons performed by this block are listed in Table 3.

For pin OUTA, the compare registers CMPA1 and CMPA2 define three different intervals. The value driven onto the OUTA pin can be specified by the host driver for each of the three intervals. For pin OUTB, the compare registers CMPB1 and CMPB2 perform a similar function.

In addition to the interval comparisons, exact match comparisons between the counter and each of the four compare registers is performed. While not implemented in this design, the HSC N2HET program is constructed with interrupt capable instructions for each comparison listed in Table 3 so that adding interrupts to the program is simply a matter of setting the interrupt request bit of the desired N2HET compare instruction.

Table 3. HSC Output Compare Block Comparisons and Actions

Comparison	Action on Pins	Interrupt Host CPU
Counter ≤ CMPA1	Drive Pin OUTA to Interval 1 Output Value	optional
CMPA1 ≤ Counter < CMPA2	Drive Pin OUTA to Interval 2 Output Value	optional
CMPA2 ≤ Counter	Drive Pin OUTA to Interval 3 Output Value	optional
CMPA1 == Counter	none	optional
CMPA2 == Counter	none	optional
Counter ≤ CMPB1	Drive Pin OUTB to Interval 1 Output Value	optional
CMPB1 ≤ Counter < CMPB2	Drive Pin OUTB to Interval 2 Output Value	optional
CMPB2 ≤ Counter	Drive Pin OUTB to Interval 3 Output Value	optional
CMPB1 == Counter	none	optional
CMPB2 == Counter	none	optional

While currently not implemented in the examples accompanying this design, any of five tests listed above can be configured to interrupt the host processor when true.

The state of each output pin can also be overridden (forced) by a signal from the Auxiliary input block. For each pin there is a force signal, and the pin state while the force is applied can be programmed to be either high or low.

The output compare function is bypassed whenever the counter state becomes invalid (saturated). In this case, unless a force signal from the auxiliary input block is applied, the output pins retain their last valid state.

3.1.4 HSC Input Capture Block

The input capture block is perhaps the simplest. Upon receiving a trigger from the Auxiliary input block, each capture register simply copies the counter value and holds it until the next trigger. The host side driver can read the capture register to determine the position at which the trigger event occurred. The capture is evaluated each loop before the counter is incremented or decremented.

3.2 N2HET Based Pulse Train Output

Figure 10 is a block diagram of the PTO function. The PTO function consists of three main functional blocks:

- Command Buffer
- Command Processor
- Step Execution

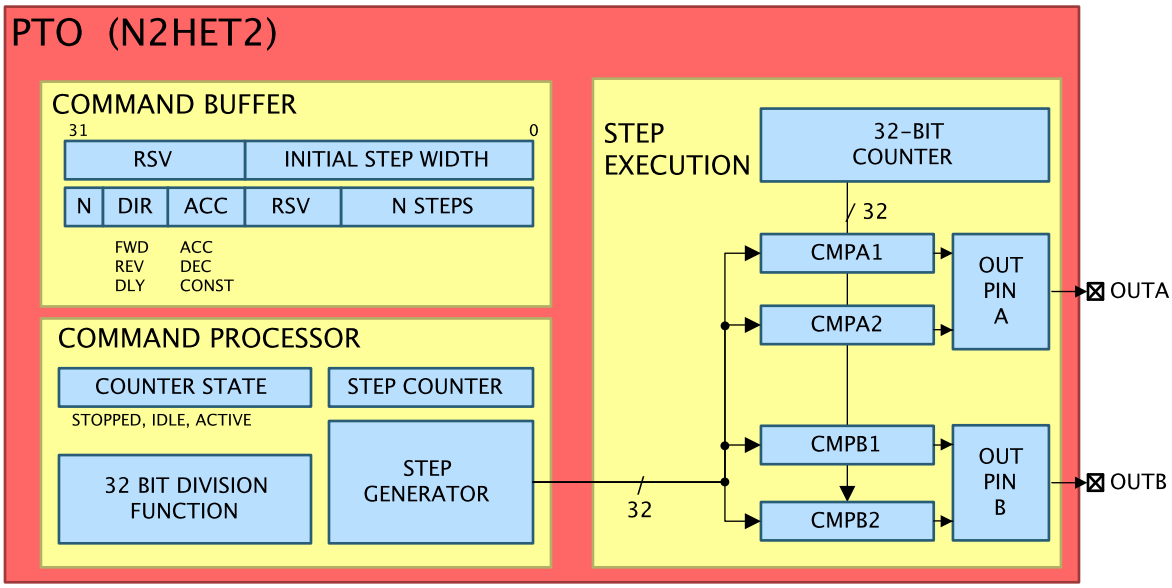


Figure 10. PTO Block Diagram

3.2.1 PTO Command Buffer Block

The PTO Command buffer consists of two 32-bit locations in the N2HET RAM where the application code submits a new command to the PTO.

Each command is made up of two 32-bit words as illustrated in Figure 10. The first 32-bit word simply contains the initial step width - measured in N2HET HR resolution clock periods (9.09ns for this example). The second 32-bit word contains fields for the number of steps to execute, the acceleration type (acceleration, deceleration, or constant speed), the direction (forward, reverse, or pure time delay with no movement in either direction), and an "N" field that indicates that the command is new and not yet processed by the N2HET. When N field is set, the host CPU should not submit another command to the PTO as this will overwrite and may corrupt the previous command that was submitted. Once the N bit is cleared, this means the N2HET has extracted the information needed from the command buffer and the CPU is free to write a new command to the command buffer.

3.2.2 PTO Command Processor Block

The PTO command processor block is responsible for keeping track of the PTO state, decoding incoming commands, and computing the width of each step in the series described by the command.

The PTO state can be either:

1. Reset - This is the initial state of the PTO. In this state the PTO drives its outputs to a predefined state (Low,Low) and ignores any commands submitted to the command buffer.
2. Idle - The prior command completed and the command buffer was empty. In this state the PTO maintains the last state on the OUTA and OUTB pins and continually checks the submission of a new command to the command buffer. When a new command is detected the PTO state transitions from Idle back to Active.
3. Active - Currently Executing a Prior Command. A new command may be submitted to the command buffer and will begin execution as soon as the current command completes.

These three states as illustrated by Figure 11.

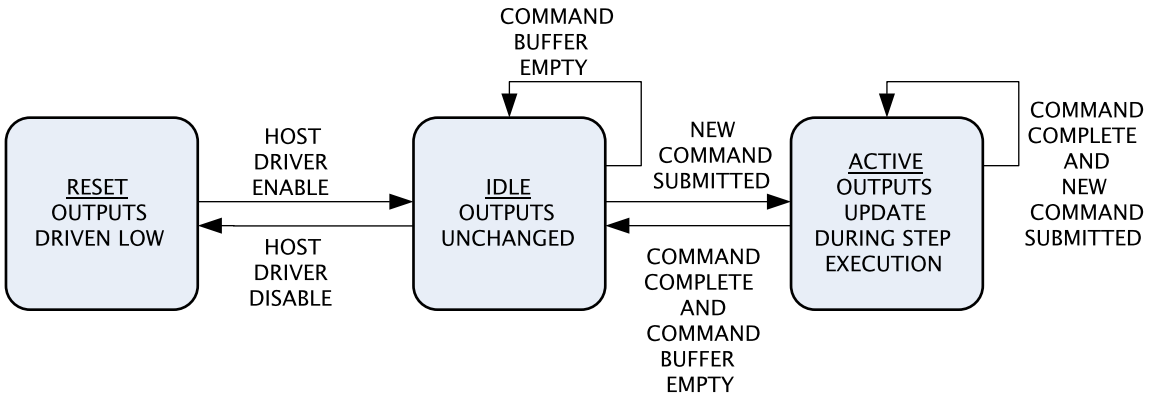


Figure 11. State Transitions of the PTO Command Processor Block

3.2.3 PTO Step Execution Block

The PTO Step execution block simply consists a 32-bit free running counter and of a pair of output compare registers for each output pin. With two output compares per pin, each pin can be toggled up to twice during a given step period. For example in the Count/Dir mode, CMPA1 and CMPA2 are used to first drive the 'Count' signal high, then drive it low to create a pulse.

Figure 12 illustrates how the PTO command processor block configures the four output compare registers to create a simple Count/Dir Output waveform. The first compare point CMPB1 is used to set OUTB (direction) to indicate the current direction. A fixed delay after this compare point, a rising edge on OUTA is created by CMPA1. The delay provides setup time for the direction signal before the rising edge of the count signal. Approximately one half of the pulse width later, CMPA2 is configured to drive OUTA low again to complete the generation of the COUNT pulse high period. Finally the CMPB2 pin is used to time the end of the current step and to begin processing of any subsequent steps.

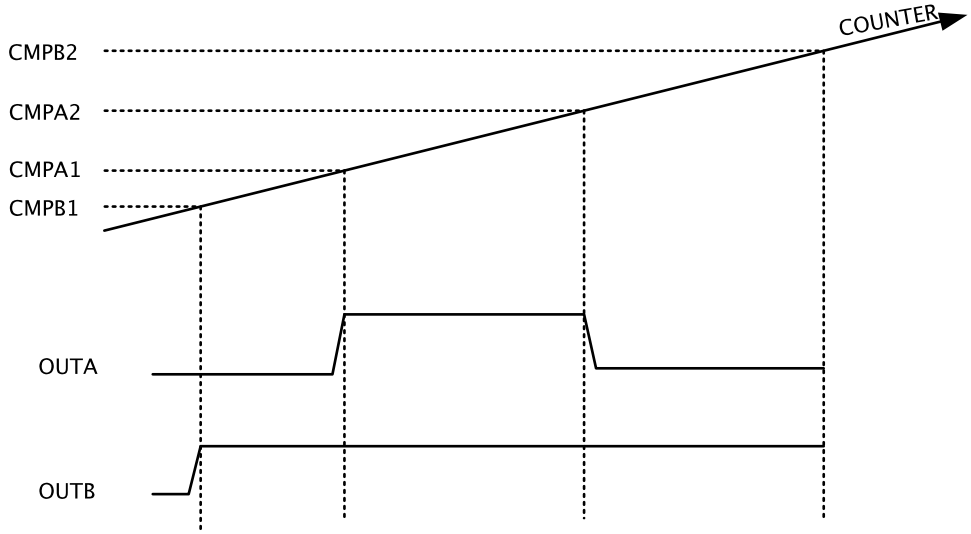


Figure 12. Illustration of how the two output compares of the step execution block are used to create a Count/Dir Waveform

4 System Design Theory

This section describes the design of the HET code for the HSC and PTO functions.

4.1 HSC Design

This section describes the design and implementation of the HSC N2HET program and host-side driver functions. The specific example of the HSC quadrature mode is used but the same theory applies to the other HSC modes.

4.1.1 HSC Counter Block Implementation

The HW filtering step of the HSC counter block uses the N2HET hardware suppression filtering feature. The HW suppression filter allows the application to set a minimum pulse duration which will be allowed to pass through the filter for processing by the N2HET. Filtering is controlled by registers HETSFPRLD and HETSFENA. The hardware suppression filter feature is chosen for inputs INA, INB because these inputs are expected to have a higher operating frequency than the auxiliary inputs and the filter allows the minimum pulse width to be configured with resolution up to the N2HET bus clock frequency. The operation of the filter HW suppression filter function is described in theRM57Lx Technical Reference Manual [8].

After filtering the inputs INA and INB, the signals need to be decoded to determine when to increment or decrement the counter. The decoding portion of the N2HET HSC code is different for each of the three counting modes. The code for Quadrature Mode is shown in Example 1. It simply tests for an active edge first on INA,, then on INB. Once an active edge is detected, the code checks if the edge is rising or falling. Once that is determined, the code checks the level of the other counter input pin to determine whether to branch to H1_CNT_UP or H1_CNT_DN. If no edges are detected, then the code branches to H1_CNT_END as the counter will not be updated during the current loop resolution period.

Example 1. HSC Counter Decoding - Quadrature Mode

```
;-------------------------------------------------------------------------------------------
; H1 COUNTER DECODING
;
; Count Up   if:
; (A Rise && B Low)  || (A Fall && B High) || (B Rise && A High) || (B Fall && A Low)
;
; Count Down if:
; (A Rise && B High) || (A Fall && B Low)  || (B Rise && A Low)  || (B Fall && A High)
;
; For 2x Mode, change H1_CNTB1 to always branch to H1_CNT_END
; For 1x Mode, make the same change plus change H1_CNTA4 to always branch to H1_CNT_END
;-------------------------------------------------------------------------------------------
H1_CNTA1   BR {event = BOTH, pin=PIN_H1_INA, cond_addr = H1_CNTA2,  next=H1_CNTB1}
H1_CNTA2   BR {event = RISE, pin=PIN_H1_INA, cond_addr = H1_CNTA3,  next=H1_CNTA4}
H1_CNTA3   BR {event = LOW,  pin=PIN_H1_INB, cond_addr = H1_CNT_UP, next = H1_CNT_DN}
H1_CNTA4   BR {event = LOW,  pin=PIN_H1_INB, cond_addr = H1_CNT_DN, next = H1_CNT_UP}

H1_CNTB1   BR {event = BOTH, pin=PIN_H1_INB, cond_addr = H1_CNTB2,  next=H1_CNT_END}
H1_CNTB2   BR {event = RISE, pin=PIN_H1_INB, cond_addr = H1_CNTB3,  next=H1_CNTB4}
H1_CNTB3   BR {event = LOW,  pin=PIN_H1_INA, cond_addr = H1_CNT_DN, next = H1_CNT_UP}
H1_CNTB4   BR {event = LOW,  pin=PIN_H1_INA, cond_addr = H1_CNT_UP, next = H1_CNT_DN}
```

When the initial decoding produces a count up or a count down signal, the next step is to apply Hysteresis in case of a direction change. The code to implement Hysteresis is shown in Example 2. This code simply resets the hysteresis count of the opposite counting direction, then decrements the hysteresis counter for the current counting direction skipping the counter code when the hysteresis count is nonzero.

Example 2. HSC Counter Hysteresis

```
;-------------------------------------------------------------------------------------------
; H1 COUNTER HYSTERESIS  - DOWN DIRECTION
;-------------------------------------------------------------------------------------------
```

Example 2. HSC Counter Hysteresis (continued)

```
H1_CNT_DN  MOV32 {type=IMTOREG&REM, reg=NONE, remote=H1_HYS_UP, data = H1_CNT_HYSV_LR }
H1_HYS_DN  DJZ   {next=H1_CNT_END, cond_addr = H1_CNT_DEC, reg=NONE, data=H1_CNT_HYSV_LR }

;-------------------------------------------------------------------------------------
; H1 COUNTER HYSTERESIS - UP   DIRECTION
;-------------------------------------------------------------------------------------
H1_CNT_UP  MOV32 {type=IMTOREG&REM, reg=NONE, remote=H1_HYS_DN, data = H1_CNT_HYSV_LR }
H1_HYS_UP  DJZ   {next=H1_CNT_END, cond_addr = H1_CNT_INC, reg=NONE, data=H1_CNT_HYSV_LR }
```

After applying Hysteresis, if the counter needs to be incremented or decremented then this action is performed. The code that performs these functions is shown in Example 3

First a simple +1 or -1 operation is performed on the current count (stored in register T). Then the updated count value is checked to see if it crosses the counter minimum or maximum limit. If so then either the rollover or saturation operation is applied. The default code loaded into the N2HET implements rollover, but by changing the next address field of the H1_CNT_RST the operation can be changed to saturation.

After these checks have been applied, if the counter value changes it is written to H1_CNT_VALUE so that the count value can be fetched by other functional blocks.

Example 3. HSC Counter Saturation or Rollover and Count Value / Valid Bit

```
;-------------------------------------------------------------------------------------
; H1 COUNT WITH SATURATION/ROLLOVER - DOWN
;-------------------------------------------------------------------------------------
H1_CNT_DEC      SUB   {src1=T,   src2=IMM,  dest=S,    data=0, hr_data=1}
H1_CNT_NRS_REG  ADD   {src1=IMM, src2=ZERO, dest=R,
                       data=H1_CNT_DRSV_LR, hr_data=H1_CNT_DRSV_HR}
H1_CNT_MIN_REG  SUB   {src1=T,   src2=IMM,  dest=NONE,
                       data=H1_CNT_MINV_LR, hr_data=H1_CNT_MINV_HR, next=H1_CNT_RS1}
;-------------------------------------------------------------------------------------
; H1 COUNT WITH SATURATION/ROLLOVER - UP
;-------------------------------------------------------------------------------------
H1_CNT_INC      ADD   {src1=T, src2=IMM, dest=S, data=0, hr_data=1}
H1_CNT_PRS_REG  ADD   {src1=IMM, src2=ZERO, dest=R,
                       data=H1_CNT_URSV_LR, hr_data=H1_CNT_URSV_HR}
H1_CNT_MAX_REG  SUB   {src1=T,   src2=IMM,  dest=NONE,
                       data=H1_CNT_MAXV_LR, hr_data=H1_CNT_MAXV_HR, next=H1_CNT_RS1}
;-------------------------------------------------------------------------------------
; H1 COUNT WITH SATURATION/ROLLOVER COMMON
;   Driver software must change next address of H1_CNT_RS2 to:
;     - H1_CNT_END     for rollover mode
;     - H1_CNT_VALID for saturation mode
;-------------------------------------------------------------------------------------
H1_CNT_RS1      BR    {event=NZ, cond_addr=H1_CNT_VALUE}
H1_CNT_RS2      ADD   {src1=R, src2=ZERO, dest=NONE, rdest=REM, remote=H1_CNT_VALUE,
                       data=0, hr_data=0, next=H1_CNT_END}
;-------------------------------------------------------------------------------------
; H1 COUNTER VALUE
;-------------------------------------------------------------------------------------
H1_CNT_VALUE    ADD   {src1=S, src2=ZERO, dest=IMM, data=0, hr_data=0, next=H1_CNT_END}

;-------------------------------------------------------------------------------------
; H1 COUNTER VALID
;-------------------------------------------------------------------------------------
H1_CNT_VALID    ADD {src1=ZERO, src2=ZERO, dest=IMM, data=0, hr_data=0, next=H1_CNT_END}
```

The counter enable, sync, and reset functions are implemented prior to processing of INA, INB. The code for these functions is found in Example 4. Each function is gated by an OR-AND combination of the debounced auxiliary inputs and software triggers based on an event vector held within register B. The event detection is described in greater detail in Section 4.1.2. The logical 'OR' of the event inputs is implemented with the AND (bitwise-AND) instruction followed by a BR (branch) on non-zero result. For example, for the counter reset function the AND operation at H1_RES_Q1T followed by the BR operation at H1_RES_Q1B will continue on to test the second 'OR' condition (H1_RES_Q2T, H1_RES_Q2B) if any one OR more of the edge detection bits in the register B match a corresponding bit that is set in the qualifier bit mask held in the data field of H1_RES_Q1T.

Note that because at least one of the four conditions for each pin (Rising Edge, High, Falling Edge, Low) must be true, a qualifier mask that contains all four bits set will match unconditionally. This feature is used when only a single 'OR' condition is desired or when continuous triggering is needed.

For the counter reset and sync functions, when the qualifier tests pass, the counter is simply loaded with a preset value. For the reset function this is performed by the H1_RES_REG. For sync this is performed by H1_SYN_REG. In addition, the sync operation also sets the counter valid state to valid by H1_SYN_VAL.

The same OR-AND event qualifier is used to decode the count enable, but the result is simply to either skip the remainder of the counting block or to execute it.

Example 4. HSC Counter Enable, Sync and Reset Functions

```
;-------------------------------------------------------------------------------------------
; H1 COUNTER RESET
;-------------------------------------------------------------------------------------------
H1_RES_Q1T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_RES_Q1V, hr_data=0}
H1_RES_Q1B      BR    {event=NZ, cond_addr=H1_RES_Q2T, next=H1_RES_END}
H1_RES_Q2T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_RES_Q2V, hr_data=0}
H1_RES_Q2B      BR    {event=NZ, cond_addr=H1_RES_REG, next=H1_RES_END}
H1_RES_REG      ADD   {src1=IMM, src2=ZERO, dest=NONE, rdest=REM, remote=H1_CNT_VALUE,
                       data=H1_CNT_RESV_LR, hr_data=H1_CNT_RESV_HR}
H1_RES_END
;-------------------------------------------------------------------------------------------
; H1 COUNTER SYNC
;-------------------------------------------------------------------------------------------
H1_SYN_Q1T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_SYN_Q1V, hr_data=0}
H1_SYN_Q1B      BR    {event=NZ, cond_addr=H1_SYN_Q2T, next=H1_SYN_END}
H1_SYN_Q2T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_SYN_Q2V, hr_data=0}
H1_SYN_Q2B      BR    {event=NZ, cond_addr=H1_SYN_REG, next=H1_SYN_END}
H1_SYN_REG      ADD   {src1=IMM, src2=ZERO, dest=NONE, rdest=REM, remote=H1_CNT_VALUE,
                       data=H1_CNT_SYNV_LR, hr_data=H1_CNT_SYNV_HR}
H1_SYN_VAL      ADD   {src1=IMM, src2=ZERO, dest=NONE, rdest=REM, remote=H1_CNT_VALID,
                       data=1, hr_data=0}
H1_SYN_END
;-------------------------------------------------------------------------------------------
; H1 COUNTER ENABLE
;-------------------------------------------------------------------------------------------
H1_ENA_Q1T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_ENA_Q1V, hr_data=0}
H1_ENA_Q1B      BR    {event=NZ, cond_addr=H1_ENA_Q2T, next=H1_OUTPUTS}
H1_ENA_Q2T      AND   {src1=IMM, src2=B, dest=NONE, data=H1_ENA_Q2V, hr_data=0}
H1_ENA_Q2B      BR    {event=NZ, cond_addr=H1_ENA_CNTR, next=H1_OUTPUTS}
H1_ENA_CNTR
```

4.1.2 HSC Auxiliary Input Block Implementation

The auxiliary input block is responsible for the debounce and edge/level detection on the slow auxiliary inputs INW, INX, INY, INZ.

The debounce algorithm is the same algorithm that is implemented by the N2HET hardware suppression filter logic, except that the auxiliary inputs are sampled once per loop resolution period (1.16 µs for this design) and they are debounced in software. An example of the debounce of pin INW is shown in Example 5.

Example 5. HSC Debounce Code for Auxiliary Input A

```
;-------------------------------------------------------------------------------------------
; H1 DEBOUNCE PIN_H1_INW
; Steps:
;    1. Reset debounce count when any edge is detected on PIN_H1_INW
;       1a.  Detect any edge (rise or fall) on PIN_H1_INW
;       1b.  Reset debounce count to max value if edge is detected
;    2. Skip to end (H1_INW_DEBE) and decrement debounce count - if pin is not yet debounced
;    3. If pin is debounced (counter reached 0 before entering into step 2)
;       3a. Set a bit mask in Reg A indicating a valid sample on pin during this loop.
;       3b. Set bit in register B if pin is high
; NB: for PIN_H1_INW, bit position in register R,S is determined by H1_INW_MSK
;-------------------------------------------------------------------------------------------
H1_INW_DEB1 BR   {event=BOTH, pin=PIN_H1_INW, cond_addr=H1_INW_DEB2, next=H1_INW_DEB3}
H1_INW_DEB2 MOV32{type=IMTOREG&REM, reg=NONE, remote=H1_INW_DEB3, data=H1_INW_DEBV,hr_data=0}
H1_INW_DEB3 DJZ  {cond_addr=H1_INW_DEB4, next=H1_INW_DEBE, data=H1_INW_DEBV}
H1_INW_DEB4 AND  {src1=A, src2=IMM, dest=A, data=H1_INW_NMSK, hr_data=0}
H1_INW_DEB5 BR   {event=HIGH, pin=PIN_H1_INW, cond_addr=H1_INW_DEB6, next=H1_INW_DEBE}
H1_INW_DEB6 OR   {src1=B, src2=IMM, dest=B, data=H1_INW_MSK,  hr_data=0, next=H1_INW_DEBE}
H1_INW_DEBE
```

After each of the four pins is debounced, the results are accumulated. Two results are computed. First, the cumulative debounce status of each of the auxiliary inputs is tracked. The HSC is programmed to wait until all of the auxiliary inputs have been debounced at least one time and therefore have a valid initial state before any processing is performed.

Second a vector that contains bit fields representing the detection of a Rising Edge, Falling Edge, High Level (excluding rising edge), and Low Level (excluding falling edge) is detected during the current loop resolution period. This vector is used by the other functional blocks to trigger operations like the count enable and reset. The code that implements the debounce accumulate function and produces the event vector is shown and explained in Example 6

Example 6. HSC Debounce Accumulate Code

```
;-------------------------------------------------------------------------------------------
; H1 DEBOUNCE ACCUMULATE
;-------------------------------------------------------------------------------------------
; Accumulate Debounced Pin Status of HSC H1 Auxiliary Inputs for use by Input Actions
; Upon Entry
; Register A - the negative bit mask for pins that have debounced values in the current loop.
; Register B - values of pins that are debounced, 0 for pins not debounced in current loop.
; The bit masks are of the form [P P P P] where P = [W X Y Z].  (Repeated 4 times)
; The repetition facilitates the calculation of edge/level detection.
; The Data field of H1_CUM_DEB1 is initially [Fh Fh Fh Fh]
;
; Steps:
;    1.  AND (to clear bits) H1_CUM_DEB1 with Register A.
;        When H1_CUM_DEB1 == 0 all pins have been debounced at least once.
;    2.  Skip to the end of H1 until all inputs are debounced.
;    3.  Fetch previous debounced pin state to Register R
;    4.  Extract the unchanged pin values from Register R into Register S
;    5.  Combine newly debounced pin states with previous pin states to compute the
;        current debounced pin state into register B
;    6.  Load the previous debounced pin state from H1_DEB_RSLT into register A
;    7.  Store the current debounced pin state from register B into H1_DEB_RSLT for next loop
;    8.  Register A format: [P0 P0 P0 P0] where P0 = [W0 X0 Y0 Z0] - the previous debounced
;        pin state.  XOR A with a bit mask to transform its contents to [P0 !P0 P0 !P0]
;    9.  Register B format: [P1 P1 P1 P1] where P1 = [W1 X1 Y1 Z1] - the current debounced
;        pin state.  XOR B with a bit mask to transform its contents to [!P1 !P1 P1 P1]
;    10. Logically AND B and A and store the result into B and also H1_PIN_EVTS
;        The result is:
;           [P0 !P0 P0 !P0] * [!P1 !P1 P1 P1] = [P0*!P1 !P0*!P1 P0*P1 !P0*P1]
```

Example 6. HSC Debounce Accumulate Code (continued)

```
;        which provides the following information about the debounced pin states:
;            P0*!P1  indicates falling edges on pins W X Y Z
;           !P0*!P1  indicates pins W X Y Z low but not falling
;            P0* P1  indicates W X Y Z high but not rising
;           !P0* P1  indicates rising edges on pins W X Y Z
;
;    11. The above steps 3-11 must run twice before the result is valid.
;        Skip to the end of H1 after only the initial run.
;
;-----------------------------------------------------------------------------------------
H1_CUM_DEB1 AND  {src1=IMM, src2=A, dest=IMM, data=H1_ALL_MSK, hr_data=0}
H1_CUM_DEB2 BR   {next=H1_END, event=Z,  cond_addr=H1_CUM_DEB3}
H1_CUM_DEB3 AND  {src1=REM, src2=A, dest=A, data=0, hr_data=0, remote=H1_DEB_RSLT}
H1_CUM_DEB4 XOR  {src1=REM, src2=IMM, dest=R, data=H1_PRV_MSK, hr_data=0, remote=H1_DEB_RSLT}
H1_DEB_RSLT OR   {src1=A, src2=B, dest=B, data=0, hr_data=0, remote=H1_DEB_RSLT, rdest=REM}
H1_CUM_DEB5 XOR  {src1=IMM, src2=B, dest=B, data=H1_CUR_MSK, hr_data=0}
H1_PIN_EVTS AND  {src1=R, src2=B, dest=B, data=0, hr_data=0, remote=H1_PIN_EVTS, rdest=REM}
H1_CUM_DEB6 DJZ  {next=H1_END, data=1, cond_addr=H1_SW_TRIG}
;-----------------------------------------------------------------------------------------
; Software Triggers Both Pulse & Persistent
;-----------------------------------------------------------------------------------------
H1_SW_TRIG       OR   {src1=B, src2=IMM,  dest=B,    data=0,            hr_data=0}
H1_SW_TRIG_SAV1  AND  {src1=B, src2=IMM,  dest=A,    data=H1_SWCLR_CLR, hr_data=0}
H1_SW_TRIG_SAV2  XOR  {src1=A, src2=ONES, dest=NONE, rdest=REM,
                       remote=H1_END, data=0, hr_data=0}
```

4.1.3 HSC Output Compare Block Implementation

The output compare block first checks the counter valid state to determine if the compare operations should be skipped. Assuming the counter is valid, the next step is to check the current count value against the two compare registers for each output pin.

The code for pin OUTA is shown in Example 7. Note that two tests are performed on each compare value. First an exact match to the compare register is tested, (ex H1_CMPA_INT1). Then a test for less than or equal to is performed (H1_CMPA_TST1).

Either of these two tests could be programmed to generate a compare match interrupt, simply by turning on the interrupt enable field of the BR instruction. This design does not make use of that capability but it is assumed that it will be required and therefore the code makes the provision for the exact match interrupt even if it is not needed to determine the output pin state.

After the counter is tested against both compare registers, the range is known and the appropriate output value for that range is stored in register R (for OUTA).

Example 7. HSC Output Compare Code Example for OUTA

```
;-----------------------------------------------------------------------------------------
; H1 OUTPUT COMPARE - OA
;-----------------------------------------------------------------------------------------
H1_CMPA_CMP1    SUB   {src1=IMM, src2=T, dest=NONE, data=H1_CMPA_THR1, hr_data=0}
H1_CMPA_INT1    BR    {event=Z,  cond_addr=H1_CMPA_TST1, next=H1_CMPA_TST1}
H1_CMPA_TST1    BR    {event=LO, cond_addr=H1_CMPA_CAC1, next=H1_CMPA_CMP2}
H1_CMPA_CAC1    MOV32 {type=IMTOREG, reg=R, data=H1_CMPA_ACT1_LR, hr_data=H1_CMPA_ACT1_HR,
                       next=H1_CMPB_CMP1}
H1_CMPA_CMP2    SUB   {src1=IMM, src2=T, dest=NONE, data=H1_CMPA_THR2, hr_data=0}
H1_CMPA_INT2    BR    {event=Z,  cond_addr=H1_CMPA_TST2, next=H1_CMPA_TST2}
H1_CMPA_TST2    BR    {event=LO, cond_addr=H1_CMPA_CAC2, next=H1_CMPA_CAC3}
H1_CMPA_CAC2    MOV32 {type=IMTOREG, reg=R, data=H1_CMPA_ACT2_LR, hr_data=H1_CMPA_ACT2_HR,
                       next=H1_CMPB_CMP1}
H1_CMPA_CAC3    MOV32 {type=IMTOREG, reg=R, data=H1_CMPA_ACT3_LR, hr_data=H1_CMPA_ACT3_HR,
                       next=H1_CMPB_CMP1}
```

After the compare operation, the force output action is checked. If the force output condition is triggered, then the force output value is shifted out onto the pin. If not, then the output from the compare register is pushed onto the pin if the compare was tested. If the compare was skipped, then the pin is left unchanged.

The code that implements this function for pin OUTA is shown in Example 8. Note that the compare values for drive high and drive low are both non-zero, so a zero condition at H1_OUTA_T2 indicates that the compare was skipped as the compare block always initializes register R to zero prior to testing the counter valid bit.

Example 8. HSC OUTA code prioritizing force over compare functions

```
;-------------------------------------------------------------------------------------------
; H1 OUTPUT A
;   Force Conditions Prioritized over Compare Match
;-------------------------------------------------------------------------------------------
H1_FRCA_Q1T     AND   {src1=IMM, src2=B, dest=NONE, data=H1_FRCA_Q1V, hr_data=0}
H1_FRCA_Q1B     BR    {event=Z, cond_addr=H1_OUTA_T1}
H1_FRCA_Q2T     AND   {src1=IMM, src2=B, dest=NONE, data=H1_FRCA_Q2V, hr_data=0}
H1_FRCA_Q2B     BR    {event=Z, cond_addr=H1_OUTA_T1}
H1_FRCA_CAC     MOV32 {type=IMTOREG,  reg=R, data=H1_FRCA_ACT_LR, hr_data=H1_FRCA_ACT_HR}
H1_OUTA_T1      ADD   {src1=R, src2=ZERO, dest=NONE, data=0, hr_data=0, rdest=REM,
                       remote=H1_OUTA_ACT}
H1_OUTA_T2      BR    {event=Z, cond_addr=H1_OUTA_END, next=H1_OUTA_ACT}
H1_OUTA_ACT     SHFT  {smode=OR1, cond=UNC, pin=PIN_H1_OA, data=0, cond_addr=H1_OUTA_END}
H1_OUTA_END
```

4.1.4 HSC Input Capture Block Implementation

The input capture block is very simple, it consists of a qualifier sequence plus a copy of the counter value into the capture register if both qualification conditions pass. The code that implements Input Capture A is shown in Example 8. The capture code executes in each loop prior to processing the counter so that the capture is performed on the count value that was present when the edge was asserted.

Example 9. HSC Input Capture A Code Example

```
;-------------------------------------------------------------------------------------------
; H1 CAPTURE A OPERATION
;-------------------------------------------------------------------------------------------
H1_CAPA_Q1T     AND   {src1=IMM, src2=B, dest=NONE, data=H1_CAPA_Q1V, hr_data=0}
H1_CAPA_Q1B     BR    {event=NZ, cond_addr=H1_CAPA_Q2T, next = H1_CAPA_END}
H1_CAPA_Q2T     AND   {src1=IMM, src2=B, dest=NONE, data=H1_CAPA_Q2V, hr_data=0}
H1_CAPA_Q2B     BR    {event=NZ, cond_addr=H1_CAPA_REG, next = H1_CAPA_END}
H1_CAPA_REG     ADD   {src1=REM, src2=ZERO, dest=IMM, remote=H1_CNT_VALUE, data=0, hr_data=0}
H1_CAPA_END
```

4.1.5 HSC Host Side Driver Implementation

The HSC Host Side driver is packaged as a library project. This project is located in ***<install dir>\ccs_proj\hsc_common*** (See Section 6 for an explanation of the firmware installation process and directory structure).

When including the HSC function in your application you may include the compiled library hsc.lib at link time and include the header file hsc.h in your application code. The library includes four different N2HET programs (one for each HSC mode plus a 'null' program to put the N2HET into an idle state until the HSC function is initialized).

4.2 PTO Design

This section describes the design and implementation of the PTO N2HET program and host-side driver functions. The specific example of the PTO count/dir mode is used but the same theory applies to the other PTO modes.

4.2.1 PTO Design Equations

The PTO function uses the equations described in *Generate stepper-motor speed profiles in real time* [1] to compute linear acceleration and deceleration profiles efficiently using Taylor series approximations to the ideal equations.

Equation 1 is the formula used to approximate the next pulse width in a series from the prior pulse width during acceleration, and Equation 2 is the formula used during deceleration where m is the total number of steps in the series.

$$c_i = c_{i-1} - \frac{2c_{i-1}}{4n_i + 1}, n_i = i, i = 1, 2, \ldots \quad (1)$$

$$c_i = c_{i-1} - \frac{2c_{i-1}}{4(m - n_i) + 1}, n_i = i, i = 1, 2, \ldots \quad (2)$$

For the N2HET implementation of the PTO, we have implemented a division function to compute the difference between subsequent pulses. The numerator of the division operation is initialized to twice the value shown in Equation 1 and Equation 2 because after the division is complete the result is rounded by adding 1 and dividing by 2.

The formula for the division numerator is shown in Equation 3

$$N_i = 4C_{i-1} \quad (3)$$

The denominator is initialized to the value D_1 according to Equation 4 if accelerating, or Equation 5 if decelerating. Based on the initial denominator value, each subsequent denominator in the series can be quickly calculated by either adding or subtracting 4 from the previous denominator.

$$D_i = \begin{cases} 5 & i = 1 \\ D_{i-1} + 4 & i = 2, 3, \ldots \end{cases} \quad (4)$$

$$D_i = \begin{cases} 4m - 5 & i = 1 \\ D_{i-1} - 4 & i = 2, 3, \ldots \end{cases} \quad (5)$$

4.2.2 PTO Command Buffer Block Implementation

The N2HET code for the PTO Command buffer block, as well as the step counter (part of the PTO Command Processor) is shown in Example 10.

The host side driver should write each command to the data field of the instructions P1_BUF0 and P1_BUF1, in that order, when it needs to issue a new command to the PTO function.

When the PTO command processor determines that all steps of the current command are complete (based on the test performed by the instruction P1_CUR_STEP_COUNT), the command buffer is evaluated again. After executing the instructions P1_BUF0 and P1_BUF1, the command buffer contents are copied to the data field of the instruction P1_NEXT_PW_A and to the register S respectively.

If P1_CHKCMD_A determines that there is not a valid command in the command buffer, then control moves to P1_IDL_C and the PTO function enters the idle state, because the previous command completed and there is no new command to begin processing.

On the other hand, if P1_CHKCMD_A determines that a new command has been issued then P1_BUF1 is cleared to signal to the host side driver that the buffer is empty and another command may be issued. The new command will continue to be decoded from the copy of P1_BUF1 previously stored in register S.

Instructions P1_CHKCMD_B through P1_CHKCMD_E unpack the other fields of P1_BUF1 (from register S) and move the relevant information to other points in the program where this information is required. For example, P1_CHKCMD_C unpacks the step count field and copies it to P1_CUR_STEP_COUNT

The acceleration field is tested by P1_CHKCMD_F and P1_CHKCMD_G to determine whether an acceleration, deceleration, or constant speed series was requested by the command. The calculations for each of these three cases are different, and therefore execution either goes to P1_ACCCMD_A, P1_DECCMD_A, or to P1_CSTCMD_C to seed the initial pulse width computation values to match the type of acceleration requested.

After seeding the initial pulse width computation values, control is passed back to P1_CUR_STEP_COUNT which should now contain a non-zero value. This will cause the actual execution of the first step to begin with the branch to P1_NEXT_PW_A.

Example 10. PTO Command Buffer Code

```
; PTO is Active - Check for Command
P1_CUR_STEP_COUNT    DJZ {cond_addr=P1_BUF0, next=P1_NEXT_PW_A, reg=A, data=0}

;-------------------------------------------------------------------------------------------
; Command Buffer - written by CPU or DMA, 2 32-bit words:
;
;
; P1_BUF0        31      28  27                                                          0
;                  Rsvd - 0s                                      Initial Period
;
; P1_BUF1        31      28  27          24   23         22  21 20 19                   0
;                     New      Fwd / Rev/ Dly   Dec/Acc/Const  Resvd         Pulse Count
;                  Cmd=xxx1    0001 1111 0000   11  10  00      00            1 to 1M
;
;-------------------------------------------------------------------------------------------
P1_BUF0       ADD   {src1=ZERO, src2=IMM, dest=NONE, data=0, hr_data=0,
                     rdest=REM, remote=P1_NEXT_PW_A}
P1_BUF1       ADD   {src1=ZERO, src2=IMM, dest=S, data=0, hr_data=0, smode=csl, scount=4}

; Interrupt Generating Instruction for Next Command
P1_CHKCMD_A   BR    {event=C, cond_addr=P1_BUFCLR, next=P1_IDL_C}
P1_BUFCLR     MOV32 {type=IMTOREG&REM, reg=NONE, remote=P1_BUF1, data=0, hr_data=0}
P1_CHKCMD_B   AND   {src1=IMM, src2=S, dest=NONE, data=P1_DIRMASK_LR, hr_data=P1_DIRMASK_HR,
                     smode=ASR, scount=28, rdest=REM, remote=P1_CUR_DIR}
P1_CHKCMD_C   AND   {src1=IMM, src2=S, dest=R,    data=P1_CNTMASK_LR, hr_data=P1_CNTMASK_HR,
                     smode=LSL, scount=3,  rdest=REM, remote=P1_CUR_STEP_COUNT}
P1_CHKCMD_E   AND   {src1=IMM, src2=S, dest=NONE, data=P1_ACCMASK_LR, hr_data=P1_ACCMASK_HR,
                     smode=CSL, scount=5}
P1_CHKCMD_F   BR    {event=C, cond_addr=P1_CHKCMD_G,  next=P1_CSTCMD_C}
P1_CHKCMD_G   BR    {event=N, cond_addr=P1_DECCMD_A,  next=P1_ACCCMD_A}

;Acceleration
P1_ACCCMD_A   ADD   {src1=ZERO, src2=IMM,  dest=NONE, data=0, hr_data=1,
                     rdest=REM, remote=P1_NEXT_PW_K}
P1_ACCCMD_B   ADD   {src1=ZERO, src2=IMM,  dest=NONE, data=0, hr_data=4,
                     rdest=REM, remote=P1_NEXT_PW_J}
P1_ACCCMD_C   MOV64 {cntl_val=P1_CVAL_SUB_REM_R_R_REM, data=0, hr_data=0,
                     remote=P1_NEXT_PW_M, next=P1_CUR_STEP_COUNT}
;Deceleration
P1_DECCMD_A   SUB   {src1=R,    src2=IMM,  dest=NONE, data=0, hr_data=P1_ONE_FOURTH_HR,
                     rdest=REM, remote=P1_NEXT_PW_K, smode=ASR, scount=5}
P1_DECCMD_B   SUB   {src1=ZERO, src2=IMM,  dest=NONE, data=0, hr_data=4, rdest=REM,
                     remote=P1_NEXT_PW_J}
P1_DECCMD_C   MOV64 {cntl_val=P1_CVAL_ADD_REM_R_R_REM, data=0, hr_data=0,
                     remote=P1_NEXT_PW_M, next=P1_CUR_STEP_COUNT}
;Constant Speed
P1_CSTCMD_C   MOV64 {cntl_val=P1_CVAL_NOP, data=0, hr_data=0,
                     remote=P1_NEXT_PW_M, next=P1_CUR_STEP_COUNT}
```

NOTE: While not implemented in this design, the choice of a 'BR' instruction for P1_CHKCMD_A allows for either interrupt or DMA request generation, so that the PTO could easily execute a table of commands stored in main memory and updated either in an ISR, or by the DMA or HTU.

4.2.3 PTO Command Processor Implementation

The PTO Command processor includes code to keep track of the PTO state, the current step count, and to generate both the pattern and timing for any steps that must be executed.

The current step count was described in Section 4.2.2 because it gates the execution of the command buffer code.

The PTO state is tracked by the code shown in Example 11 which is executed after compare match of the PTO Step Execution block:

Example 11. PTO Command Processor - PTO State Code

```
;-------------------------------------------------------------------------------------------
; ECMP Should Match during this loop.
; Prepare the ECMP instructions before executing them.
; Different action depending on state:   Stepping,  Idle,  Active
; State Encoding:
; Reset  :  P1_UPD_A = 0x0000000
; Idle   :  P1_UPD_A = 0x4000000
; Active :  P1_UPD_A = 0x8000000
;-------------------------------------------------------------------------------------------
P1_UPD_A  ADD {src1=IMM, src2=ZERO, dest=NONE, data=0, hr_data=0, smode=csl, scount=1}

P1_UPD_B  BR  {event=C, cond_addr=P1_CUR_STEP_COUNT, next=P1_UPD_C}
P1_UPD_C  BR  {event=N, cond_addr=P1_IDL_A,          next=P1_RST_A}

; Reset State - Set compare for next loop resolution period, and pin state for initial state
; (PIN_P1_OA=LOW,PIN_P1_OB=LOW by default)

P1_RST_A RADM64 {en_pin_action=ON,reg=T,pin=PIN_P1_OA,action=CLEAR,data=1,hr_data=0,
                 cond_addr=P1_ECMP2,remote=P1_ECMP1,comp_mode=ECMP};
P1_RST_B RADM64 {en_pin_action=ON,reg=T,pin=PIN_P1_OB,action=CLEAR,data=1,hr_data=0,
                 cond_addr=P1_UPD_A,remote=P1_ECMP2,comp_mode=ECMP,next=P1_END};

; Idle State-Set compare for current loop resolution period but do not change the pin state.
P1_IDL_A    ADD {src1=T, src2=IMM, dest=NONE, data=1, hr_data=0, rdest=REM, remote=P1_ECMP1}
P1_IDL_B    ADD {src1=T, src2=IMM, dest=NONE, data=1, hr_data=0, rdest=REM, remote=P1_ECMP2,
                 next=P1_CUR_STEP_COUNT}
P1_IDL_C    ADD {src1=IMM, src2=ZERO, dest=NONE, data=P1_STATE_IDLE_LR,
                 hr_data=P1_STATE_IDLE_HR, rdest=REM, remote=P1_UPD_A}
P1_IDL_D    ADD {src1=T, src2=IMM, dest=NONE, data=1, hr_data=0, rdest=REM, remote=P1_ECMP1}
P1_IDL_E    ADD {src1=T, src2=IMM, dest=NONE, data=1, hr_data=0, rdest=REM, remote=P1_ECMP2,
                 next=P1_END}
```

This code executes during every loop resolution period when the PTO state is idle or in reset, because both the RADM64 instructions at P1_RST_A and P1_RST_B (reset state) and P1_IDL_A and P1_IDL_B (idle state) always set the compare instructions of the PTO Step Execution block to match the current count plus one loop resolution period (data=1 field as part of the addition).

The default state after loading the PTO code into the N2HET is reset, and in this state the PTO outputs are forced low (action=CLEAR in P1_RST_A, P1_RST_B). The host side driver needs to bring the PTO out of the reset state and into the idle state by writing a value of 0x40000000 to the data field of instruction P1_UPD_A.

In the Idle state, the PTO maintains the outputs OUTA and OUTB at their last state (high or low) by not changing the 'action' field of the ECMP instructions that are part of the step execution block. This prevents any unwanted steps from appearing on the output pins.

The PTO manages transitions from Idle to Active and vice-versa based on whether or not any commands are being executed. So after moving the PTO to the idle state, the host side driver only needs to write commands to the command buffer (P1_BUF0 and P1_BUF1 locations) to initiate pulse train execution.

The step generator code for each PTO mode is different, as the pattern of edges on outputs OUTA, OUTB differs for the count/dir, cw/ccw and quadrature modes. For this reason three variants of the PTO N2HET program are constructed and during initialization the host side driver loads the correct variant into the N2HET memory. The step generator code for the Count/Dir mode is shown in Example 12.

Example 12. PTO Step Generator Code Example - Count/Dir Mode

```
;-----------------------------------------------------------------------------------------
; COUNT/DIR
;-----------------------------------------------------------------------------------------
P1_CUR_DIR ADD {src1=REM, src2=IMM, dest=S, rdest=REM,  remote=P1_SCNT, data=00h,hr_data=0h}

P1_DIR_B   ADD {src1=REM, src2=ZERO, dest=B, rdest=NONE, data=0, hr_data=0,
                smode=lsl, scount=7, remote=P1_CUR_DIR}

P1_S1T  ADD {src1=B,   src2=ZERO, dest=NONE,    data=00h, hr_data=0}
P1_S1BR BR  {event=Z,  cond_addr=P1_S1A1, next=P1_S1A2}
P1_S1A1 RADM64 {en_pin_action=ON, reg=T, pin=PIN_P1_OA, action=CLEAR, data=2, hr_data=64,
                cond_addr=P1_ECMP3, remote=P1_ECMP1, comp_mode=ECMP, next=P1_S2T};
P1_S1A2 RADM64 {en_pin_action=ON, reg=T, pin=PIN_P1_OA, action=SET, data=2, hr_data=64,
                cond_addr=P1_ECMP3, remote=P1_ECMP1, comp_mode=ECMP, next=P1_S2T};

P1_S2T  ADD {src1=B,   src2=ZERO, dest=NONE,    data=00h, hr_data=0}

P1_S2BR  BR  {event=N,  cond_addr=P1_S2A1,   next=P1_S2BR2}
P1_S2BR2 BR  {event=Z,  cond_addr=P1_S2A3,   next=P1_S2A2}
P1_S2A1 RADM64 {en_pin_action=ON, reg=T,pin=PIN_P1_OB,action=CLEAR, data=1, hr_data=0,
                cond_addr=P1_ECMP4, remote=P1_ECMP2, comp_mode=ECMP, next=P1_S3T};
P1_S2A2 RADM64 {en_pin_action=ON, reg=T,pin=PIN_P1_OB,action=SET, data=1,hr_data=0,
                cond_addr=P1_ECMP4, remote=P1_ECMP2, comp_mode=ECMP, next=P1_S3T};
P1_S2A3 RADM64 {en_pin_action=OFF, reg=T, pin=PIN_P1_OB, action=CLEAR, data=1, hr_data=0,
                cond_addr=P1_ECMP4, remote=P1_ECMP2, comp_mode=ECMP, next=P1_S3T};

P1_S3T
P1_S3A1 MOV64 {en_pin_action=ON, reg=T,pin=PIN_P1_OA, action=CLEAR, data=0, hr_data=0,
               cond_addr=P1_ECMP3, remote=P1_ECMP3, comp_mode=ECMP, next=P1_S4T};
P1_S3A2 MOV64 {en_pin_action=ON, reg=T, pin=PIN_P1_OA, action=CLEAR, data=0, hr_data=0,
               cond_addr=P1_ECMP3, remote=P1_ECMP3, comp_mode=ECMP,  next=P1_S4T};

P1_S4T  ADD {src1=B,   src2=ZERO, dest=NONE,    data=00h, hr_data=0}
P1_S4BR BR  {event=N,  cond_addr=P1_S4A1, next=P1_S4A2}
P1_S4A1 MOV64 {en_pin_action=OFF,reg=T,pin=PIN_P1_OB,action=CLEAR,data=0,hr_data=0,
               cond_addr=P1_UPD_A,remote=P1_ECMP4,comp_mode=ECMP,next=P1_S3TM};
P1_S4A2 MOV64 {en_pin_action=OFF,reg=T,pin=PIN_P1_OB,action=CLEAR,data=0,hr_data=0,
               cond_addr=P1_UPD_A,remote=P1_ECMP4,comp_mode=ECMP,next=P1_S3TM};
P1_S3TM  ADD   {src1=R,   src2=T, dest=S,    data=0, hr_data=0, rdest=REM, remote=P1_ECMP4}
P1_SCNT  ADD   {src1=R,   src2=ZERO, dest=R, data=0, hr_data=0, smode=lsr, scount=1}
P1_S4TM  SUB   {src1=S,   src2=R, dest=NONE, data=0, hr_data=0,
                rdest=REM, remote=P1_ECMP3, next=P1_END}
;-----------------------------------------------------------------------------------------
; COUNT/DIR
;-----------------------------------------------------------------------------------------
```

The step generator code begins by determining the current direction and storing it into register B (P1_CUR_DIR and P1_DIR_B). For count/dir mode, the direction is either positive (forward), negative (reverse) or zero in the case of a pure time delay with no steps. The addition performed in the instruction P1_CUR_DIR on the remote location P1_SCNT is used in quadrature mode to cycle either forward or backward through four different states of the quadrature output. It is not used in count/dir mode or in cw/ccw mode, but in an effort to keep the code structure the same across all three modes this operation is still performed even though it does not change the output sequence.

With the direction loaded into register B, the transitions (but not the actual transition times) for four different events are computed. The code beginning with P1_S1T determines whether to initially pulse the OUTA pin high (in case the direction is non-zero, i.e. forward or reverse) or to leave the OUTA pin low (in case the direction indicates a pure time delay with no stepping action).

In a similar manner, the code beginning with P1_S2T determines whether to initially drive OUTB high (forward direction), low (reverse direction), or to do nothing to OUTB in case the command is for a pure time delay. The last condition is implemented by setting 'en_pin_action=OFF' in the instruction at P1_S2A3.

At P1_S3T no test is performed because in the middle of the pulse it is safe to always drive the OUTA pin low. If it is already low in case of a pure time delay, there will be no transition on the pin. Otherwise this event will reset the OUTA pin halfway through the pulse width so that the next step may begin with a rising edge on OUTA.

The code at P1_S4T configures a final compare, but with no pin action (en_pin_action=OFF) for the OUTB pin. This comparison is used for timing purposes; to determine when the pulse completes and when it is time to begin processing the next step. But in the count/dir mode; the 'dir' pin updates at most once per pulse period therefore in this mode the pin action is always disabled for this event.

Finally the timing of these events must be computed. The time delay of the first event on each pin is set to +2 HET loop resolution periods for the OUTA pin and +1 HET loop resolution periods for the OUTB pin. This provides approximately 1.16 µs of fixed setup time between the direction signal and the rising out of the count pulse for the receiver. The setup time can be increased if needed by adjusting the data values at P1_S1A1, and P1_S1A2.

The final event is always scheduled with an offset of the computed pulse width (P1_S3TM). The event that drives the OUTA pin low is then scheduled for an offset of half of the period so that the duty cycle on the COUNT output stays approximately at 50%. This is handled by P1_SCNT (divide by 2) and P1_S4TM (subtracts half the pulse width from the computed end of the pulse and sets the third event time based on this value.

The step generator code just described is executed with the pulse width loaded into register R and current count value loaded into register T. The initial pulse width is extracted directly from the command buffer but subsequent pulse widths within the same command must be computed.

The current step width is computed according to the equations described in Section 4.2.1. For the division, a simple restoring division algorithm was implemented according to the algorithm described in Computer Arithmetic Algorithms [2]. Non-Restoring and signed division subroutines were also tested for the N2HET, but the acceleration gained per iteration was not enough to compensate for the additional cycles needed for the final corrections. Also the restoring division algorithm requires the fewest instructions which is important when N2HET memory resources are limited.

4.2.4 PTO Step Execution Implementation

The PTO step execution block is very simple. The code for this function is shown in Example 13

It begins with a free running counter, a counter that counts up to the maximum value of 0x1FFFFFFh and rolls over to 0x0000000.

The counter is followed by one ECMP instruction for each output pin (P1_ECMP1 and P1_ECMP2). While the block diagram shows two output comparators, the N2HET architecture has a fundamental requirement that only one high resolution instruction is executed per pin per loop resolution period. So the first two (in order of count value) compare points are loaded directly into the ECMP instructions for pins OUTA and OUTB. The second compare points are loaded into the MOV64 instructions at P1_ECMP3 and P1_ECMP4. These instructions act like a shadow register for the actual compare register. When the compare register matches, the MOV64 instruction updates the ECMP instruction with the compare value and type of compare (pin number, pin action) for the subsequent comparison.

Note that the range of the CNT instruction is actually 32-bits once the high-resolution structures are included, but the CNT instruction only refers to the upper 25 bits of loop count. The ECMP that follow compare the upper 25 bit loop count as well as the lower 7 bits of hi-resolution count.

Example 13. PTO Step Execution and Free Running Counter

```
;-----------------------------------------------------------------------------------------
; Free Running Counter
;-----------------------------------------------------------------------------------------
P1_TBASE    CNT   { comp=GE, reg=T, max=1FFFFFFh, data=1FFFFFFh};

;-----------------------------------------------------------------------------------------
; Counter Compare Function for pins P1_OA and P1_OB
;-----------------------------------------------------------------------------------------
P1_ECMP1  ECMP  {en_pin_action=ON,reg=T,pin=PIN_P1_OA,action=CLEAR,data=0,hr_data=0,
                 cond_addr=P1_ECMP2,hr_lr=high, next=P1_ECMP2}

P1_ECMP2  ECMP  {en_pin_action=ON,reg=T,pin=PIN_P1_OB,action=CLEAR,data=0,hr_data=0,
                 cond_addr=P1_UPD_A,hr_lr=high, next=P1_DIVST}

P1_ECMP3  MOV64 {en_pin_action=ON,reg=T,pin=PIN_P1_OA,action=CLEAR,data=0,hr_data=0,
                 cond_addr=P1_ECMP3,remote=P1_ECMP1,comp_mode=ECMP,next=P1_ECMP2}

P1_ECMP4  MOV64 {en_pin_action=ON,reg=T,pin=PIN_P1_OB,action=SET,  data=0,hr_data=0,
                 cond_addr=P1_UPD_A,remote=P1_ECMP2,comp_mode=ECMP,next=P1_DIVST}
```

4.2.5 PTO Host Side Driver Implementation

The PTO Host Side driver is packaged as a library project. This project is located in ***<install dir>\ccs_proj\pto_common*** (See Section 6 for an explanation of the firmware installation process and directory structure).

When including the PTO function in your application you may include the compiled library pto.lib at link time and use include the header file pto.h in your application code. The library includes four different N2HET programs (one for each PTO mode plus a 'null' program to put the N2HET into an idle state until the PTO function is initialized).

The file pto.c can be studied as an example of how the header file that is output by the HET assembler can be used from within accompanying host side driver to access the structures within a HET program. For example, Example 14 shows the code for the function *ptoRetVal_t ptoCmdSubmit(ptoInst_t ptoNum, ptoCmd_t cmd)*.

This function submits a PTO command by writing the two 32-bit values representing the command to the data fields of the instructions P1_BUF0 and P1_BUF1 (as explained in Section 4.2.2). The function that performs this task includes the header files output by the HET assembler and uses the data structures defined to access fields of the individual instructions symbolically.

Example 14. Host Side Driver Function for PTO Command Submission

```
#include "HL_reg_het.h"
#include "pto_null.h"
#include "pto_quadrature.h"
#include "pto_countdir.h"
#include "pto_cwccw.h"
#include "pto.h"
#include "std_nhet.h"

ptoRetVal_t ptoCmdSubmit(ptoInst_t ptoNum, ptoCmd_t cmd)
{

    volatile unsigned int *pBuf0;
    volatile unsigned int *pBuf1;

    ptoRetVal_t retval = ptoRetValOK;

    /* Mark this as a new command */
    cmd.ptoCmdBuf1 |= PTO_CMD_NEW;
```

Example 14. Host Side Driver Function for PTO Command Submission (continued)

```
    switch(ptoNum)
    {
    case pto1:
        switch (g_ptoMode[pto1])
        {
        case ptoModeDisable:
            break;
        case ptoModeCountDir:
            pBuf0 = &e_HETPROGRAM5_UN.Program5_ST.P1_BUF0_5.memory.data_word;
            pBuf1 = &e_HETPROGRAM5_UN.Program5_ST.P1_BUF1_5.memory.data_word;
            break;
        case ptoModeCwCcw:
            pBuf0 = &e_HETPROGRAM6_UN.Program6_ST.P1_BUF0_6.memory.data_word;
            pBuf1 = &e_HETPROGRAM6_UN.Program6_ST.P1_BUF1_6.memory.data_word;
            break;
        case ptoModeQuadrature:
            pBuf0 = &e_HETPROGRAM7_UN.Program7_ST.P1_BUF0_7.memory.data_word;
            pBuf1 = &e_HETPROGRAM7_UN.Program7_ST.P1_BUF1_7.memory.data_word;
            break;
        default:
            retval = ptoRetValInvalidMode;
            break;
        }
        break;
    default:
        retval = ptoRetValInvalidInstance;
        break;
    }

    while ((*pBuf1 & PTO_CMD_NEW) != 0);
    *pBuf0 = cmd.ptoCmdBuf0;
    *pBuf1 = cmd.ptoCmdBuf1;

    return retval;
}
```

4.2.6 PTO Execution of Trapezoidal Motion Profile

An example output waveform from the PTO in count/direction mode can be found in Figure 13. The time intervals captured by the logic analyzer were processed by a MATLAB® script that plots the effective frequency versus time for each pulse overlaid with simulated values for the same profile. The resulting plot is shown in Figure 14. The application code that creates this example by using the PTO library is very simple as shown in Example 15.

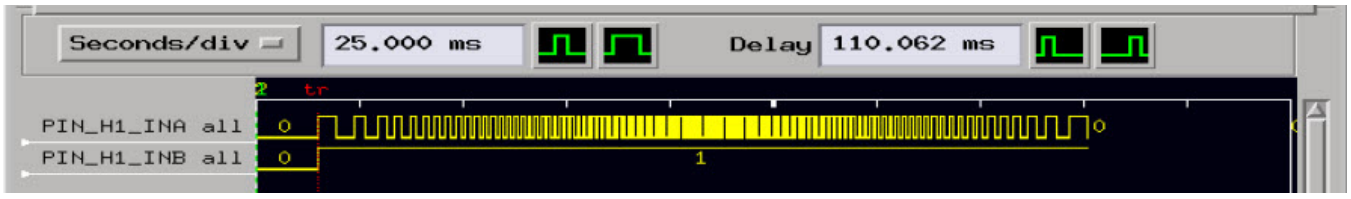


Figure 13. Example of PTO Output of Trapezoidal Motion Profile

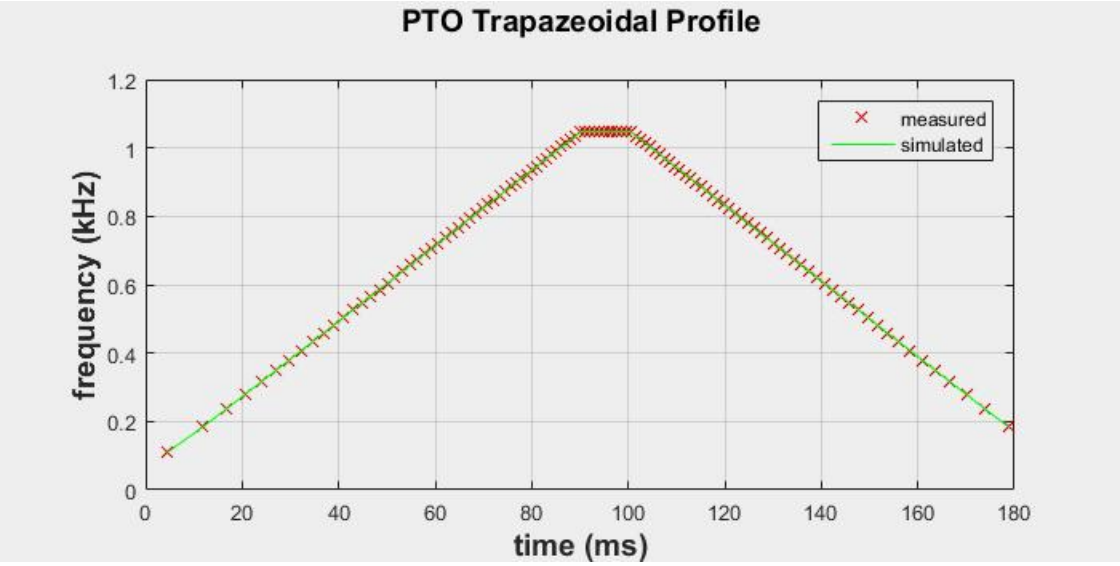


Figure 14. Plot of Measured Trapezoidal Motion Profile versus Simulated Profile

Example 15. Application Example for Creation of Trapezoidal Profile

```
/*
 * main.c
 */
#include "HL_sys_common.h"
#include "HL_het.h"
#include "pto.h"

/* Function Prototypes from pto.h for reference:
 *  ptoRetVal_t ptoInit(ptoInst_t ptoNum, ptoMode_t ptoMode);
 *  ptoRetVal_t ptoStart(ptoInst_t ptoNum);
 *  ptoRetVal_t ptoCmdSubmit(ptoInst_t ptoNum, ptoCmd_t cmd);
 *
 * ptoRetVal_t ptoCmdCreate(ptoCmd_t *cmd, uint32_t icnt, uint32_t nstp, ptoDir_t dir, ptoAcc_t acc);
 *       - icnt = initial pulse width (in counts of N2HET High Resolution Clocks
 *       - nstp = number of steps to execute in the command
 *       - dir  = direction of steps (forward, reverse, or pure time delay)
 */      - acc  = acceleration type (accelerate, decelerate, or zero acceleration/constant speed)

#define NUMCMD 3
ptoCmd_t cmdList[NUMCMD];

int main(void) {
    int I;
    hetInit();
    ptoInit(pto1, ptoModeCountDir);

    ptoCmdCreate(&cmdList[0],  1000000, 50, ptoDirFwd, ptoAccLinAcc);
    ptoCmdCreate(&cmdList[1],   105132, 10, ptoDirFwd, ptoAccZero);
    ptoCmdCreate(&cmdList[2],   105132, 50, ptoDirFwd, ptoAccLinDec);

    ptoStart(pto1);
    for (I = 0; I < NUMCMD; I++)    {
    ptoCmdSubmit(pto1, cmdList[i]);
 }
    while(1);
}
```

5 Getting Started Hardware

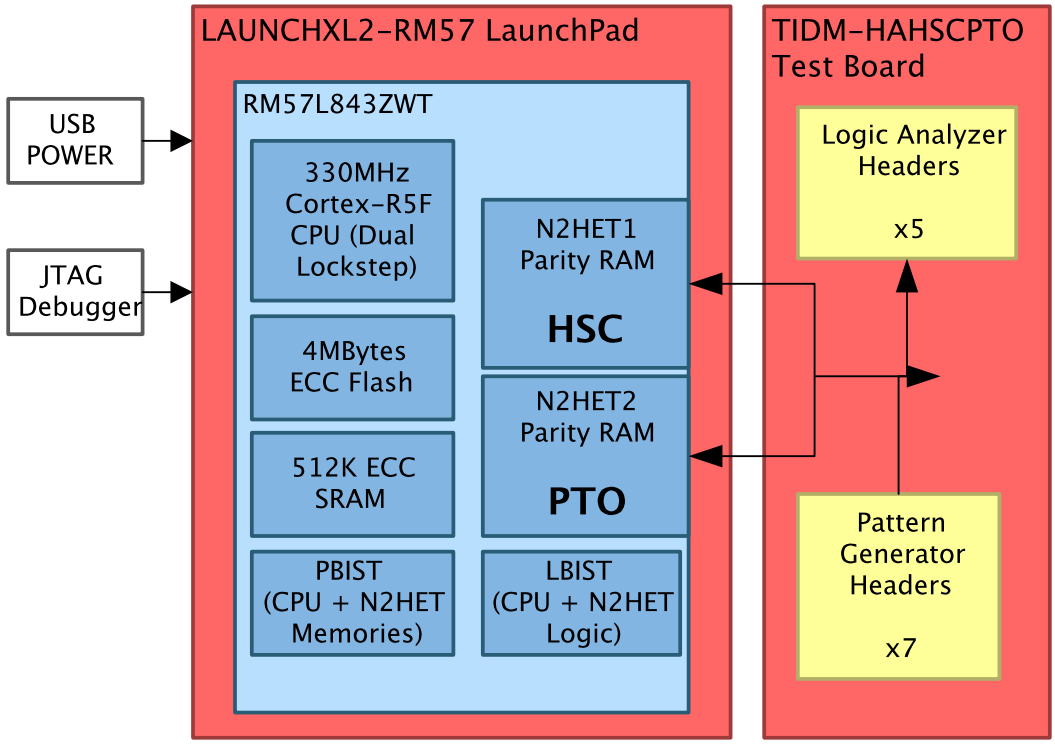


Figure 15. Hardware Block Diagram

The reference platform comprises of two different pieces of hardware as shown in Figure 15:

1. RM57L843 Launchpad - LAUNCHLX2-RM57L
2. Test Board (schematics, and design files documented in Section 9.

In addition, either jumper wires or a 20-pin ribbon cable, approximately 6" long, is needed to connect the PTO outputs to the HSC inputs in order to run the HSC tests. This is because the PTO module is used to create stimulus for the HSC module. If a 20-pin ribbon cable is used with the test board, it should have sockets on both ends. The sockets should be 2 rows, 10 positions each, with 0.100" pin to pin spacing. Suitable cables can be purchased from Mouser (517-1M-1010-020-6), Newark/Element 14 (FC20150-0), Digikey (H3CCH-2006G-ND), or many other electronic component distributors.

5.1 Hardware Setup

To set up the reference-design hardware, follow the steps listed in this section.

5.1.1 Step 1: RM57L LaunchPad Installation

Follow the instructions on the printed quick start guide that is included with the LaunchPad to setup your LaunchPad and Code Composer Studio™. You will want to verify that you can connect to the LaunchPad, download at least one example project, and debug the project with Code Composer studio before trying the application code included in this TI Design. Additional help getting started with the LaunchPad may be found on the LAUNCHXL2-RM57L Wiki.

If you only wish to run the example HSC / PTO applications without capturing the output on a logic analyzer, then you can skip the remaining steps involving the test board. However, if you chose to skip the test board steps you should make the connections outlined in Table 4 instead.

Table 4. Optional Connections: Only Make if Skipping Steps 2 - 6

HSC Signal	N2HET1 PIN	LAUNCHXL2-RM57L Locations to Jumper Across		N2HET2 Pin	PTO Signal	GIO Signal
		N2HET1	N2HET2			
INA	0	J10-20	J9-6	1	OA	-
INB	2	J4-1	J9-17	3	OB	-
INW	4	J1-9	J9-11	5	-	Out 1
INX	6	J1-3	J9-20	7	-	Out 2
INY	8	J8-3	J9-27	9	-	Out 3
INZ	10	J8-1	J9-24	11	-	Out 4
OA	1	J10-21(1)	J10-21(1)	8	-	In 1
OB	3	J10-22(1)	J10-22(1)	10	-	In 2

(1) No physical jumper required since input and output signals share the same pin.

5.1.2 Step 2 (Optional): Add Optional Sockets onto RM57L LaunchPad

The RM57L843 LaunchPad (part # LAUNCHXL2-RM57L) ships with connectors J5/J7, J8/J6, J9, and J10 unpopulated due to cost constraints. However, for this TI design these connectors must be purchased and populated. Figure 16 shows what the LaunchPad looks like from the bottom side when these sockets are added.

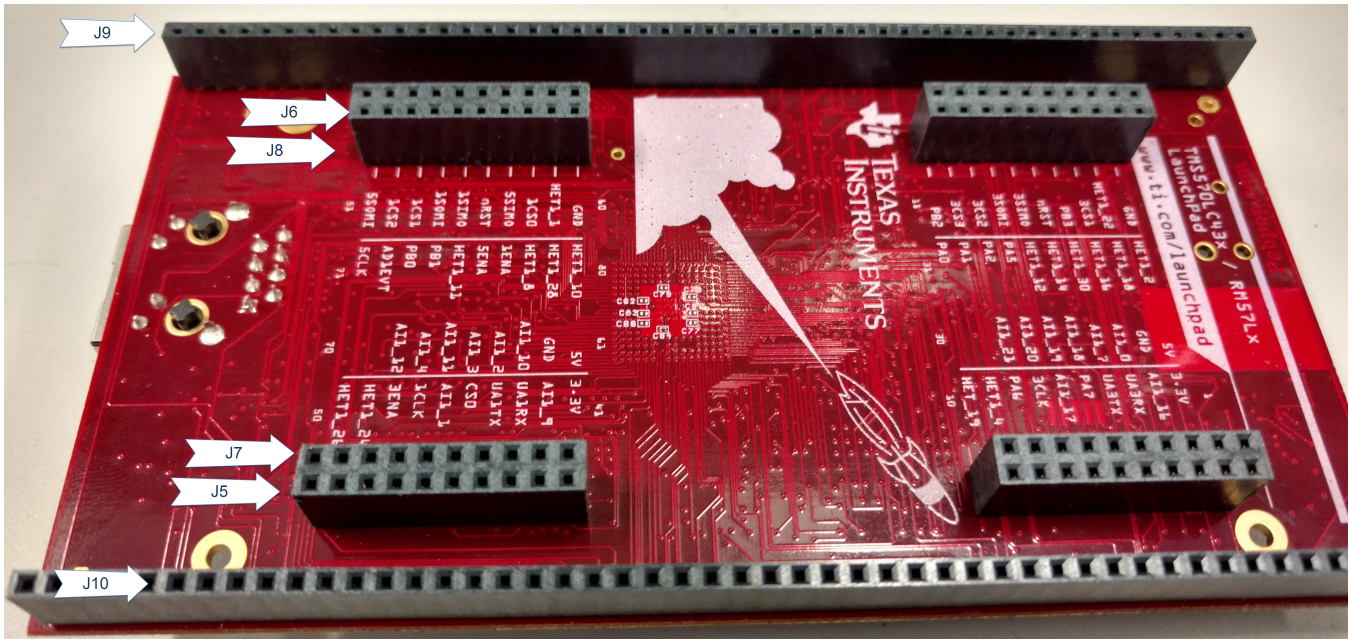


Figure 16. Bottom Side of LaunchPad with Optional Sockets Added

For J5, J6, J7, and J8, while these connectors are listed in the boards schematic as four individual 1x10 connectors with 0.100" pin spacing. However we recommend using two 2x10 connectors. One 2x10 connector can be used for J5 and J7, the other for J8 and J6. For these connector locations, a 2x10 socket should be used and assembled on the bottom side of the LaunchPad matching the assembly orientation of J1,J3 and J2,J4.

Suitable part numbers for this purpose are: Major League Electronics CRD-081413-A-F or Samtec SSQ-110-03-T-D. These receptacles have extended through-hole pin heights to match the booster pack standard and will allow stacking additional booster packs on both sides of the LaunchPad. However, this feature is not necessary for this TI design so a compatible receptacle with shorter pin lengths may be substituted if desired.

For J9, and J10, a 1x50 pin receptacle (also 0.100" pitch) is required. A suitable part number is Sullins SFH11-PBPC-D25-ST-BK. These connectors should also be assembled on the bottom side of the PCB.

5.1.3 Step 3 (Optional): Assemble Test Board

The test board should be assembled with headers J1-J10 on the top side of the board, to mate with the LaunchPad receptacles. Headers J11-J22 should be mounted on the back side of the PCB so that when the two boards are mated, logic analyzer and pattern generator pods may be plugged into J11-J22 as desired. Figure 17 shows the top side of the test board once assembled.

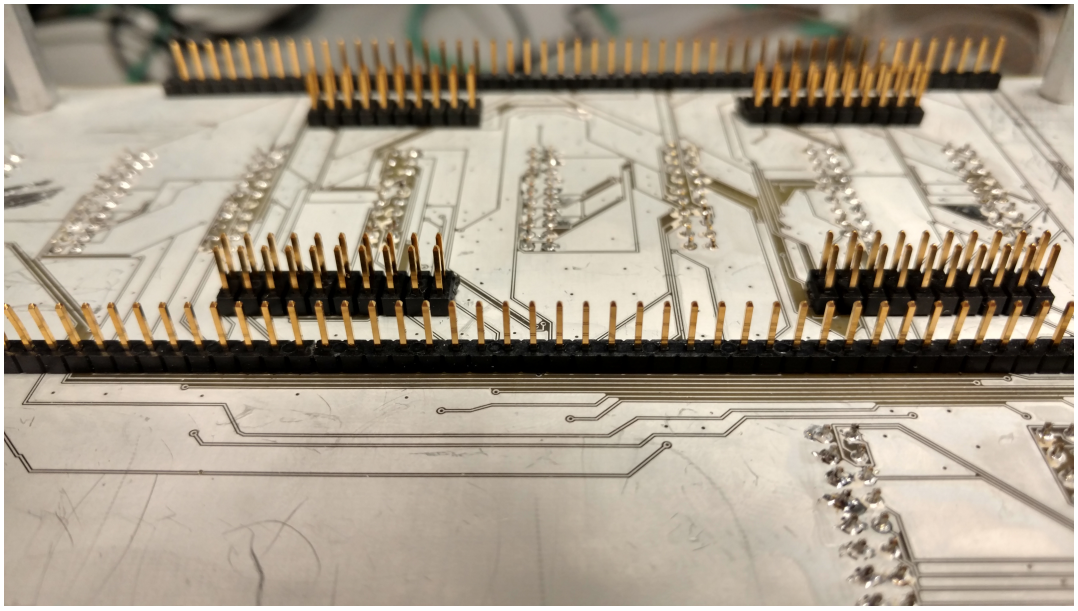


Figure 17. Top Side of Assembled Test Board

For this design, only J1-J10, J11, J12, J16 and J17 need to be populated; although populating all headers will allow the same test board to be reused for future N2HET based projects.

5.1.4 Step 4 (Optional): Connect LaunchPad and Test Board

The LaunchPad should now be mounted onto the top side of the test board. Connections should be made such that the LaunchPad J1-J10 socket mate with the test board header of the same number. For example J1 mates with J1, J2 with J2, and so on up to J10. Make sure that your orientation of the two boards matches the orientation shown in Figure 18. Also be careful to make sure that the pin 1 locations of each connector match as it is easy to be misaligned.

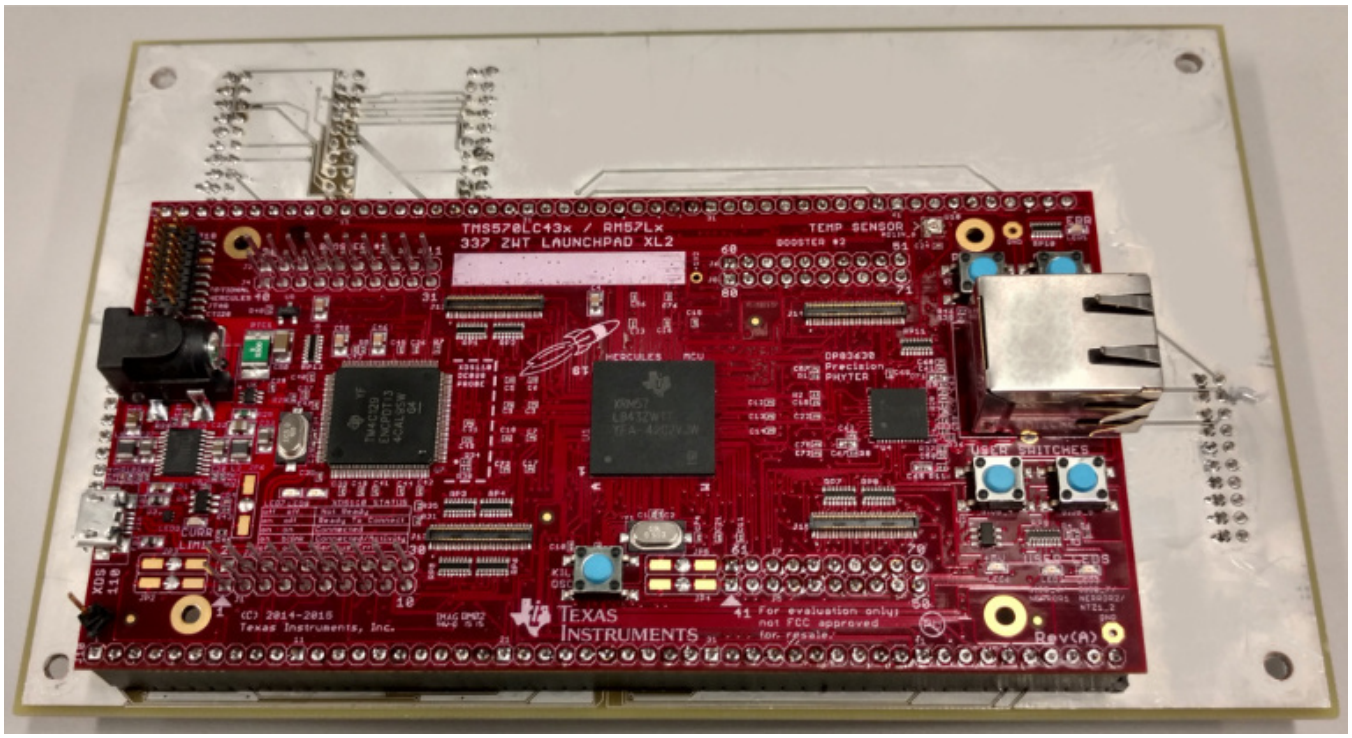


Figure 18. LaunchPad Connected to Test Board

5.1.5 Step 5 (Optional): Connect Ribbon Cable To Test Board

On the back side of the test board, connect headers J16 to J17 with a 20-pin ribbon cable or with individual jumper wires. The connections should be made straight across meaning J16-1 to J17-1, J16-2 to J17-2, etc... Figure 19 shows a ribbon cable shorting J16 and J17.

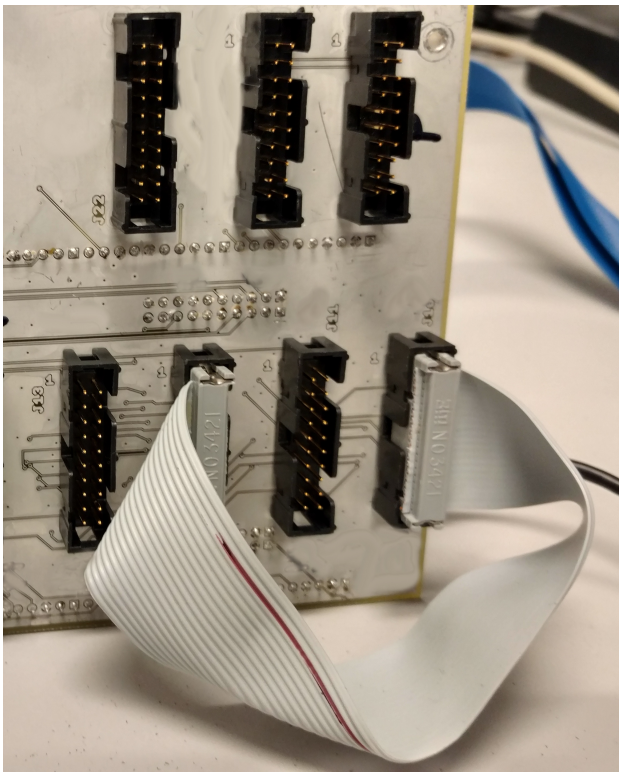


Figure 19. Ribbon Cable Plugged into Test Board J16 and J17

5.1.6 Step 6 (Optional): Connect Logic Analyzer Pods to Test Board

A logic analyzer may be connected to headers J11 and J12 on the test board in order to monitor the output of the test cases. This is optional but a logic analyzer was used to produce some of the results described in Section 8.

6 Getting Started Firmware

This section covers installing the firmware for the design on a windows machine.

To use the firmware you will need to have Code Composer Studio installed. You also will need the HET IDE. Download links for both packages are provided in Section 10. The HET IDE installs the HET assembler which is invoked during the 'clean' build target.

To rebuild the individual HET programs, you will need an environment that provides both GNU Make and the GNU M4 macro processor. For this design we used the MinGW msys 1.0 environment to provide both of these tools.

The specific version of each of these software tools that was used during the creation of this design is listed in Table 5.

Table 5. Software Tool Versions Used in Design

Tool	Version	For Additional Information
Code Composer Studio	6.1.1	http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v6
HET IDE	3.05.01	http://www.ti.com/tool/het_ide
MinGW msys	1.0	http://www.mingw.org/wiki/MSYS
GNU M4	1.4.16	http://www.gnu.org/software/m4/m4.html

6.1 Installing the Firmware

An installer is provided for this design. Download and execute the installer from TIDM-HAHSCPTO. The default installation directory for all files is ***C:\ti\Hercules\HAHSCPTO\<version>***. We will refer to the actual install folder as ***<install dir>***.

The firmware installer creates the following subdirectories under the main installation directory:

- hsc
- pto
- ccsproj

In the folder ***<install dir>\hsc\het*** there are subdirectories that contain the three versions of the HSC N2HET program:

- hsc_countdir
- hsc_cwccw
- hsc_quadrature

Additionally there are directories for the unit test of each of the hsc sub-blocks:

- debounce_test
- counter_countdir_test
- counter_cwccw_test
- counter_quadrature_test
- output_compare_test

Finally, the subdirectory:

- source

contains the macros (GNU M4 language) and a makefile that creates each of the previously listed folders. Also created are HET IDE projects that allow each program to be tested in simulation on the HET IDE.

The structure of the***<install dir>\pto\het***folder mirrors that of the hsc folder, except there are is only one unit test for the division algorithm.

The ***<install dir>\ccsproj*** folder is the starting point for work with this design. In this folder are CCS projects for each of the hsc/pto applications and for their unit tests. In the next section we will discuss how to import these projects into a CCS workspace. After you have imported these projects you can build and run the project on the test hardware.

6.2 Running the Firmware

This section explains how to import the firmware projects into Code Composer Studio and run them on the RM57L LaunchPad.

6.2.1 Step 1: Import CCS Projects Into Your Workspace

It is recommended that you create a new CCS workspace for this design because many individual projects will be imported. You can find information about creating a CCS workspace online at http://processors.wiki.ti.com/index.php/Projects_and_Build_Handbook_for_CCS#Workspaces.

Launch CCS, and make sure the Project Explorer pane is visible. From the main menu, select Project->Import CCS Projects as shown in Figure 20

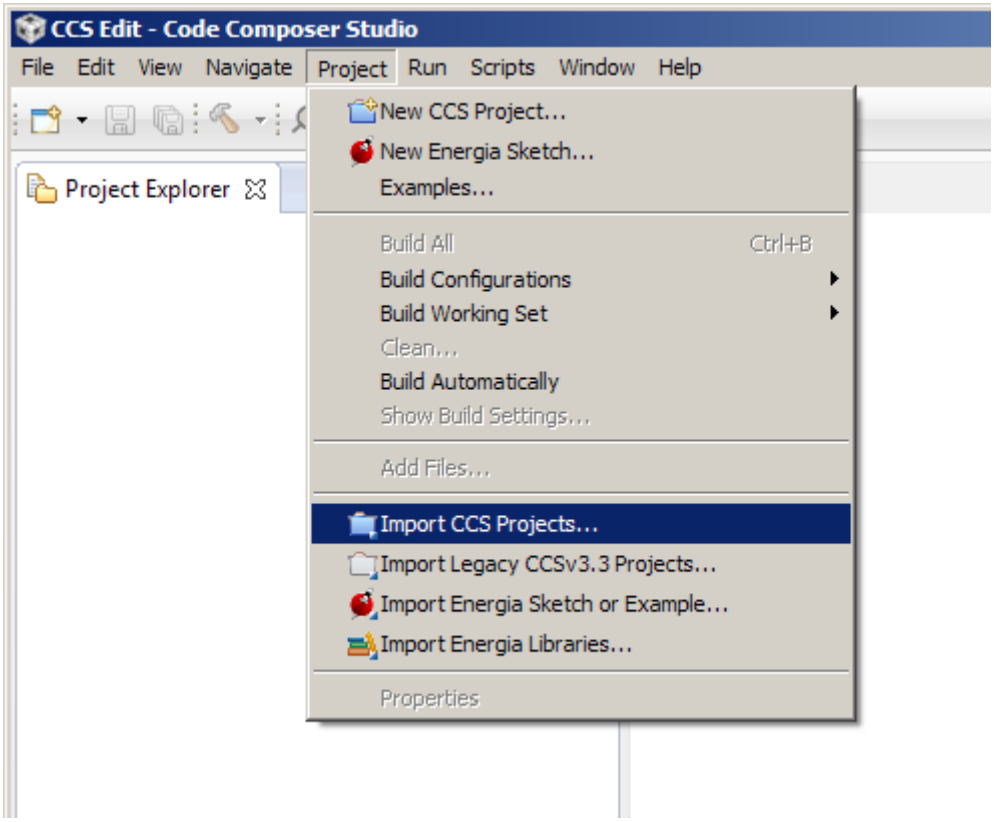


Figure 20. Import CCS Projects Menu Item

When the Select CCS Projects to Import window appears, as shown in Figure 21:

1. Chose the 'Select search-directory' option and navigate to the ***<install dir>*** folder.
2. Click on the 'Select All' button to select all the projects that CCS finds under the ***<install dir>*** folder and its subdirectories.
3. Click on the 'Finish' button and CCS will import all of the projects into the current workspace. Note that 'Copy projects into workspace' should not be selected. The sources for this project will be maintained under the ***<install dir>*** folder.

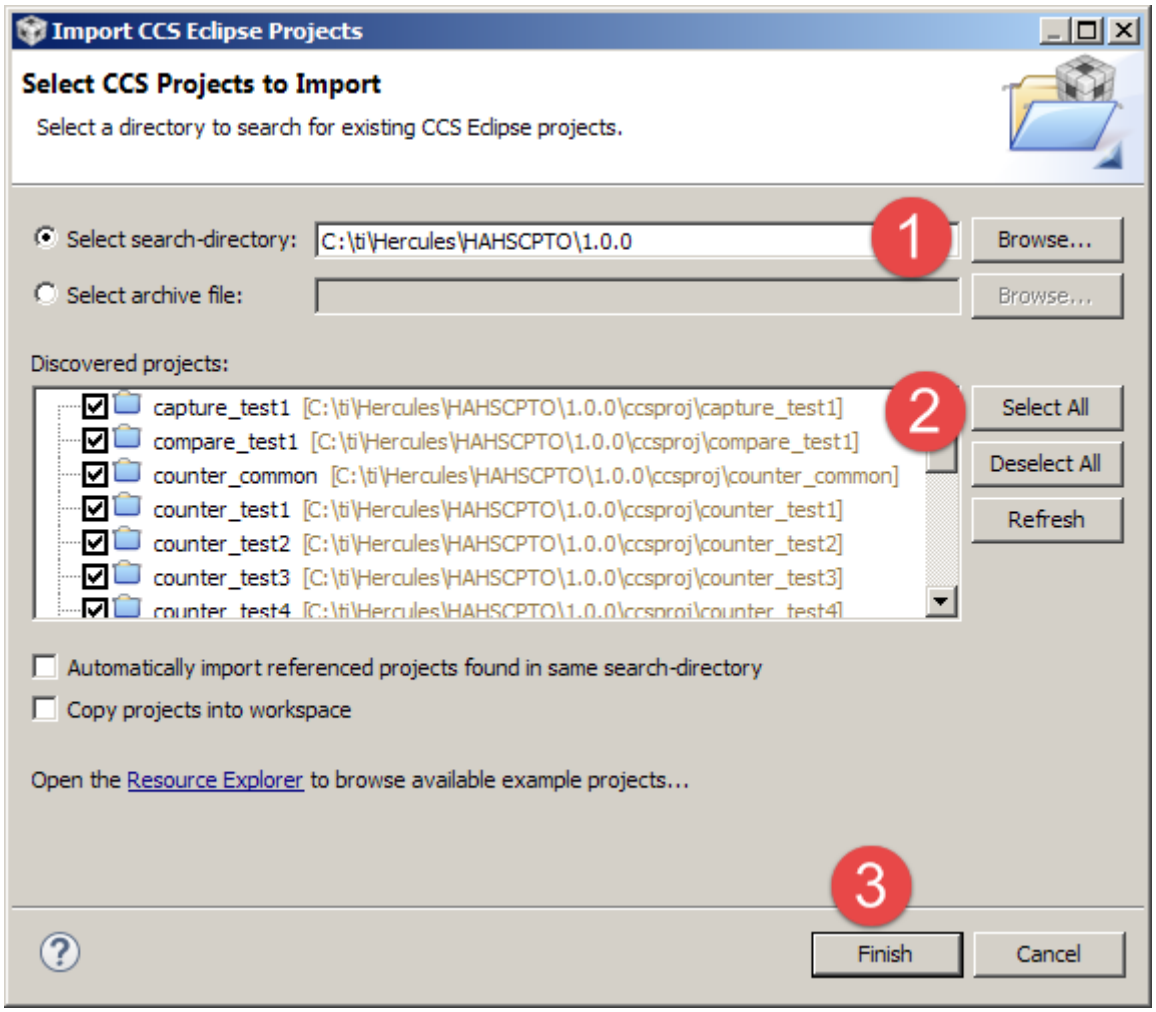


Figure 21. Importing projects from the <install dir>\ccs_projs folder.

6.2.2 Step 2: Build all of the imported projects

Next, build all of the imported projects. In the Project Explorer pane, select all of the projects then right click and select 'Build Project' from the context menu as shown in Figure 22. It may take a few minutes to complete the build, especially if the runtime support library needs to be built as well.

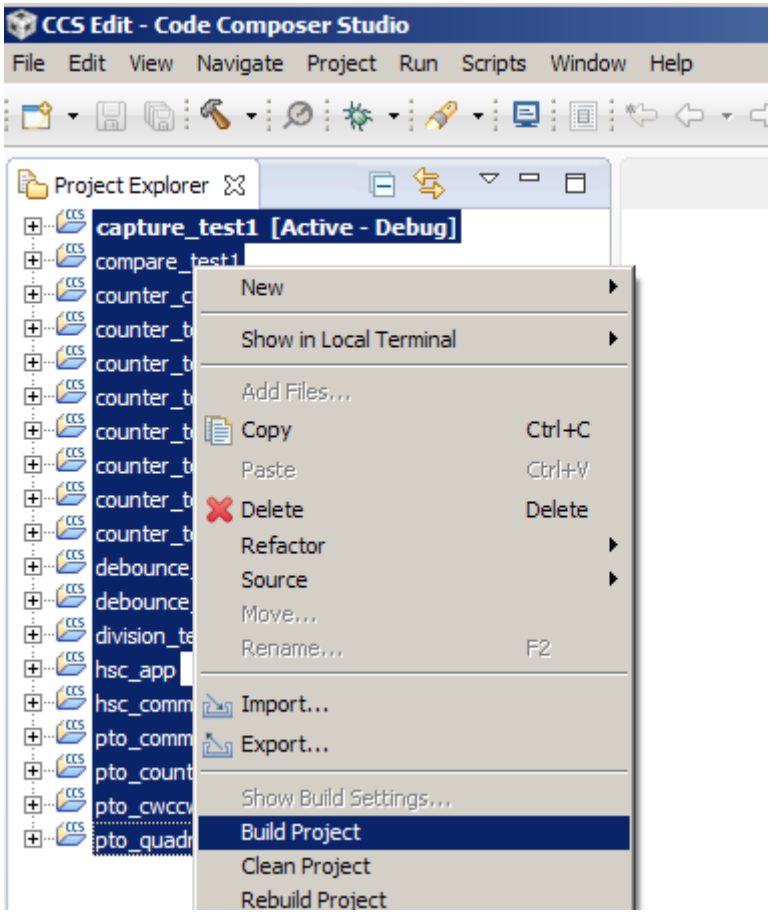


Figure 22. Build all of the imported projects

6.3 Run the individual test projects after completing the test setup

Each project should now be ready to download into the LaunchPad and execute / debug. Before doing this, please refer to Section 7 where the instructions for setting up the test hardware and a description of what each test project does can be found.

7 Test Setup

The setup to produce the test results in Section 8 is the same as described in Section 5.1. All of the optional steps involving the test board should be completed. Figure 23 is a picture of the test setup used to capture the results reported in this design guide. For this test setup, the LaunchPad can be powered through USB, no external power supply is required. However as the figure shows, an external XDS560v2 emulator was used to load the test firmware into the target. This was used strictly for the sake of speed and the convenience of the available network connection; the LaunchPad XDS110 on board emulator could be used instead of the XDS560v2 that is pictured.

The Logic Analyzer that was used to capture the test results for the PTO tests is an HP16700A series mainframe with an HP16715A 167MHz State/667MHz Timing logic analyzer card. The test board is also designed to be able to drive the N2HET pins using an HP16522A pattern generator but none of the tests developed for this design actually make use of that capability.

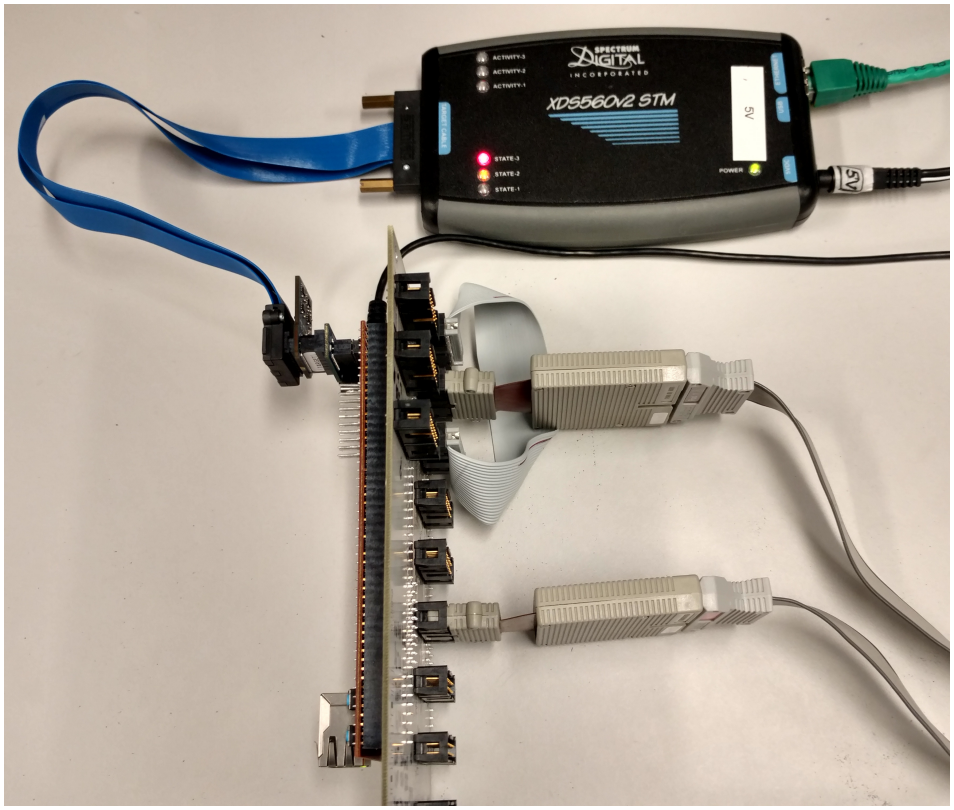


Figure 23. Test Setup

The CCS projects for each test are described in Table 6. The tests build upon each other according to the order of the table. For example first the PTO is tested, then the PTO is used to create stimulus to test the HSC. The HSC tests build up the HSC beginning with the counter tested in each counting mode, and then adding the input capture, output compare, and auxiliary input functional blocks in stages until the complete HSC application is tested.

Table 6. Test Projects

Project Name	Test Method	Targets	Description
division_test	Self Checking, Comparison against host CPU	N2HET Integer Division algorithm used by PTO to compute width of successive pulses during acceleration	Tests the 32-bit Division Algorithm used by the PTO. Same values are computed by the ARM CPU and N2HET. Number of errors are counted. Test passes when complete and err_cnt variable = 0.

Table 6. Test Projects (continued)

Project Name	Test Method	Targets	Description
debounce_test1	Logic Analyzer / Visual Check	Software Debounce of HSC Auxiliary Inputs	The event detection logic (rise, fall, high, low) for each auxiliary input is output onto pins monitored with the logic analyzer. The debounce value is set to increasingly larger values for each pin. A fixed PWM frequency is applied to all pins. The captured output is visually checked.
debounce_test2	Logic Analyzer / Visual Check	Software Debounce of HSC Auxiliary Inputs	
pto_countdir_app	Logic Analyzer / MATLAB Script	PTO Count/Dir Mode	Series of commands 10 commands with different direction, acceleration/deceleration, initial pulse width, and number of steps issued to PTO and results captured by logic analyzer. Logic analyzer results are saved to a file and processed by MATLAB script to produce a plot of the PTO output frequency versus time, as well as the expected values based on a simulation of the same commands.
pto_cwccw_app	Logic Analyzer / MATLAB Script	PTO Clockwise/Counter Clockwise Mode	
pto_quadrature_app	Logic Analyzer / MATLAB Script	PTO Quadrature Mode	
counter_test1	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Count/Dir Mode x1	Tests: Hysteresis, Counter Range, Rollover and Saturation Modes, Counter Enable, Counter Reset, Counter Sync, Counter Saturation resulting in invalid counter, Counter sync restoring valid counter status. Uses the PTO in the same mode to create stimulus and a known number of steps. Verifies that the counter value is correct after each series of steps. All triggers for Sync, Enable, Reset are issued by software. The Auxiliary Input Block, Output Compare Block, and Input Capture block are not included so that the extra N2HET RAM and IO pins can be used to output the counter value for capture on the logic analyzer (for debug purposes).
counter_test2	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Count/Dir Mode x2	
counter_test3	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Cw/Ccw Mode x1	
counter_test4	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Cw/Ccw Mode x2	
counter_test5	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Quadrature Mode x4	
counter_test6	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Quadrature Mode x2	
counter_test7	Self Checking, PTO as stimulus for HSC	HSC Counter Block - Quadrature Mode x1	
capture_test1	Self Checking, PTO as stimulus for HSC	HSC Counter + Capture Blocks, Count/Dir x1 Mode	Repeats all counter tests. Adds tests to trigger Capture A, Capture B both individually and simultaneously and checks that the capture value is correct.
compare_test1	Self Checking, PTO as stimulus for HSC	HSC Counter + Capture + Compare Blocks, Count/Dir x1 Mode	Repeats all capture_test1 tests. Adds tests for Output Compare on OUTA, OUTB. Sets up two compare registers for each pin. Steps the counter through each range forward and backward, to make sure that the output pin changes state correctly as each compare threshold is crossed.
hsc_app	Self Checking, PTO as stimulus for HSC	HSC Counter + Capture + Compare Block, and Auxiliary Input Blocks, Count/Dir x1 Mode	Complete HSC app. Removed debug output of counter value. Repeats all compare_test1 items and exercises both software triggers and triggers through auxiliary inputs.

8 Test Data

8.1 Self Checking Tests

All of the self checking tests described in Table 6 complete with an errcount = 0. These tests should be run and stopped when completed. They stop at a while(1); statement at the end of the main() function. The errcount value can be evaluated in an expression window and confirmed to be 0, as shown in Figure 24 and marked with a '1'. The while(1); loop is marked with a '2' in this figure. Of course the logic analyzer can be used to capture the test execution on the pins if a visual check is desired as well as the self check.

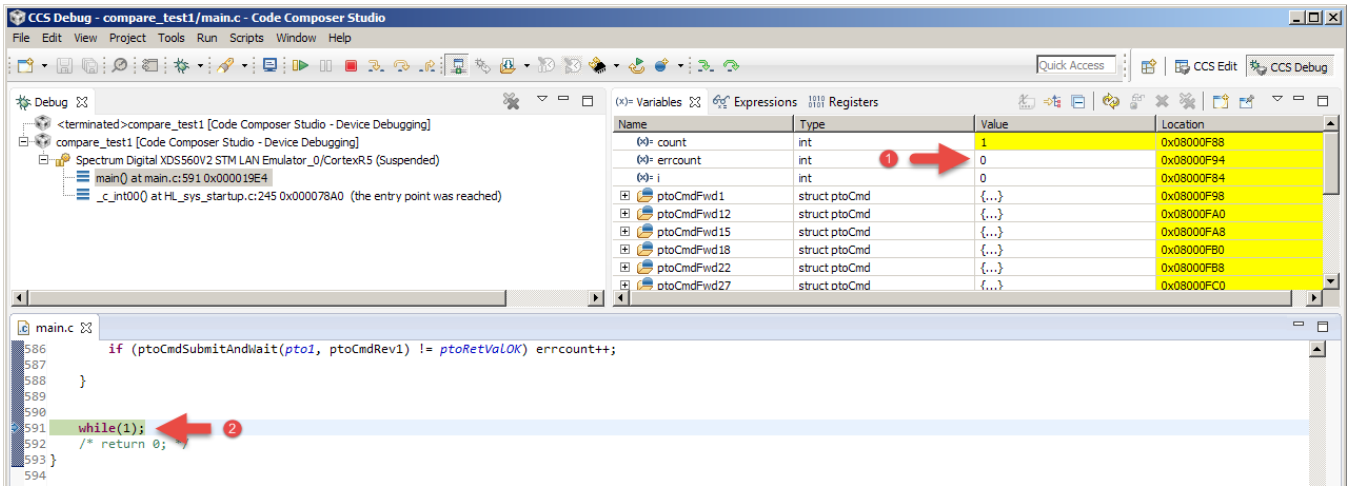


Figure 24. Successful Completion of Self-Checking compare_test1

8.2 Debounce Test Results

The tests ***debounce_test1*** and ***debounce_test2*** from Table 6 are checked visually by inspecting the logic analyzer trace results.

These tests simply debounce and run the auxiliary input edge/level detection algorithm for each pin. Then the test framework adds code to output each edge / level detection bit on a spare HET pin for monitoring purposes. Note that this extra function provided by the framework is used only for the unit test of the auxiliary inputs block and is not generated as part of the final HSC application.

Figure 25 shows the waveform generated by the project ***debounce_test1***, with arrows drawn on each waveform to correlate the rising edge of the input to the corresponding rising edge output pin. Each of the pins INW (Green), INX (Orange), INY (Purple), and INZ (Red) are plotted along with the outputs indicating detection of a rising edge (_R), high level (_H), falling edge (_F), and low level (_L) for each pin in the same color.

For ***debounce_test1***, the debounce period for pins INW, INX, INY, and INZ respectively are set to 3, 2, 1, and 0 N2HET loop resolution periods. The test input on each pin is the same waveform with a 5 N2HET loop resolution period high pulse followed by a 4 N2HET loop resolution period low pulse. In this the input pulse width is larger than the filter width for all of the input pins, but the filter delay is clearly visible and increases with increasing filter width. Also visible are the correct decoding of the Rising and Falling Edge as well as High and Low states. (High and Low were chosen to exclude the loop where the edge is detected, because the triggering equations allow the two conditions to be combined easily in the first level 'OR' logic.)

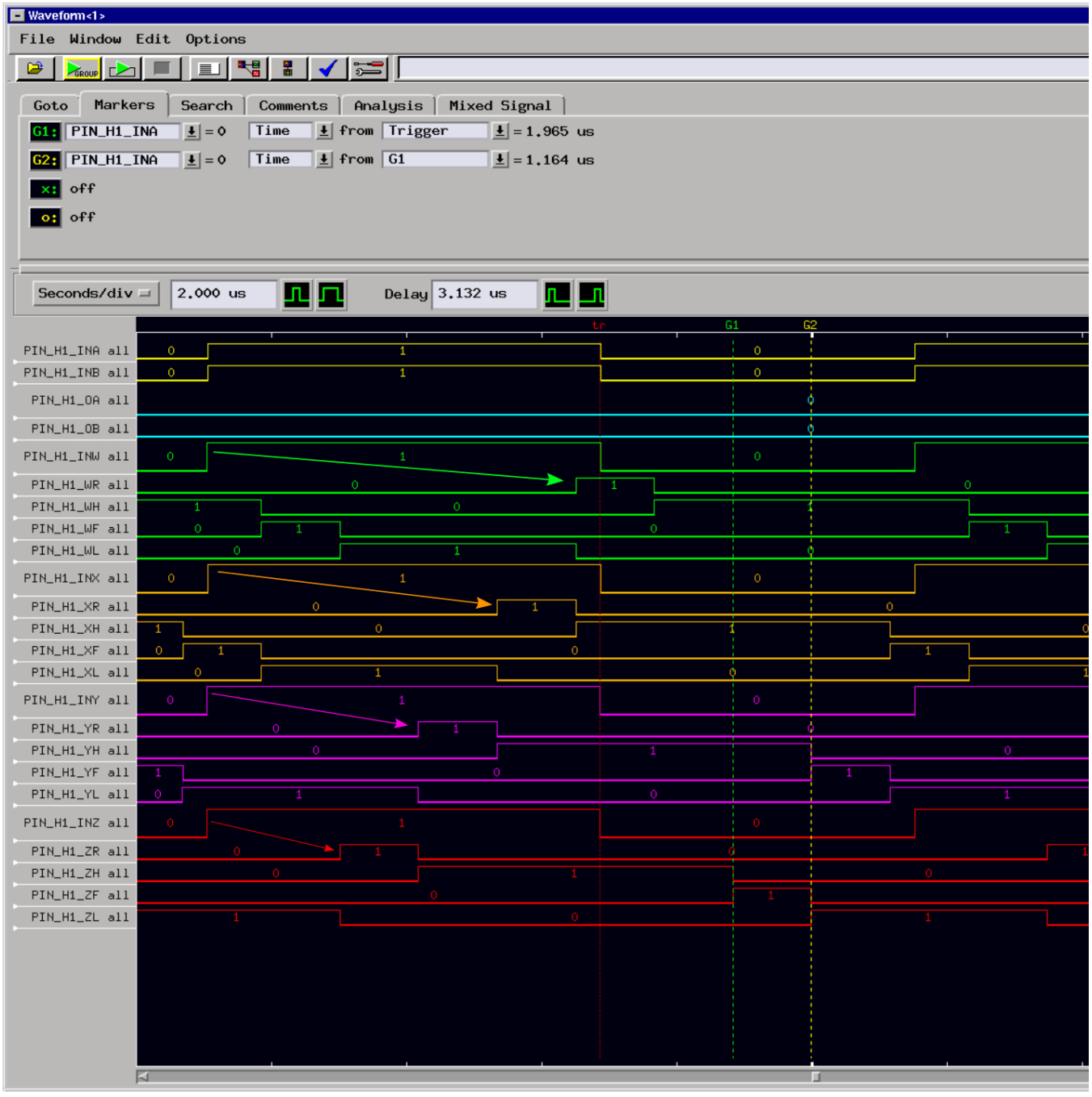


Figure 25. Debounce Test 1 Waveform

Figure 26 shows the waveform generated by the project ***debounce_test2***. For ***debounce_test1***, the debounce period for pins INW, INX, INY, and INZ respectively are set to 5, 4, 3, and 2 N2HET loop resolution periods. The input signal is the same as in ***debounce_test1***. The INW(green) result shows that both the 5 loop resolution high phase and 4 loop resolution low phase of the input signal are filtered by INW with filter value set to 5, because neither the high nor the low phase exceed the filter length. The INX(orange) signal with filter length of 4 only passes the high phase of the input pulse that is 5 cycles (1 cycle longer than the 4 cycle filter width). The low phases are completely blocked. Therefore only an initial rising edge is detected and subsequently the pin is reported as high. INY and INZ with filter lengths of 3 and 2 repetitively pass through both phases of the input signal as expected.

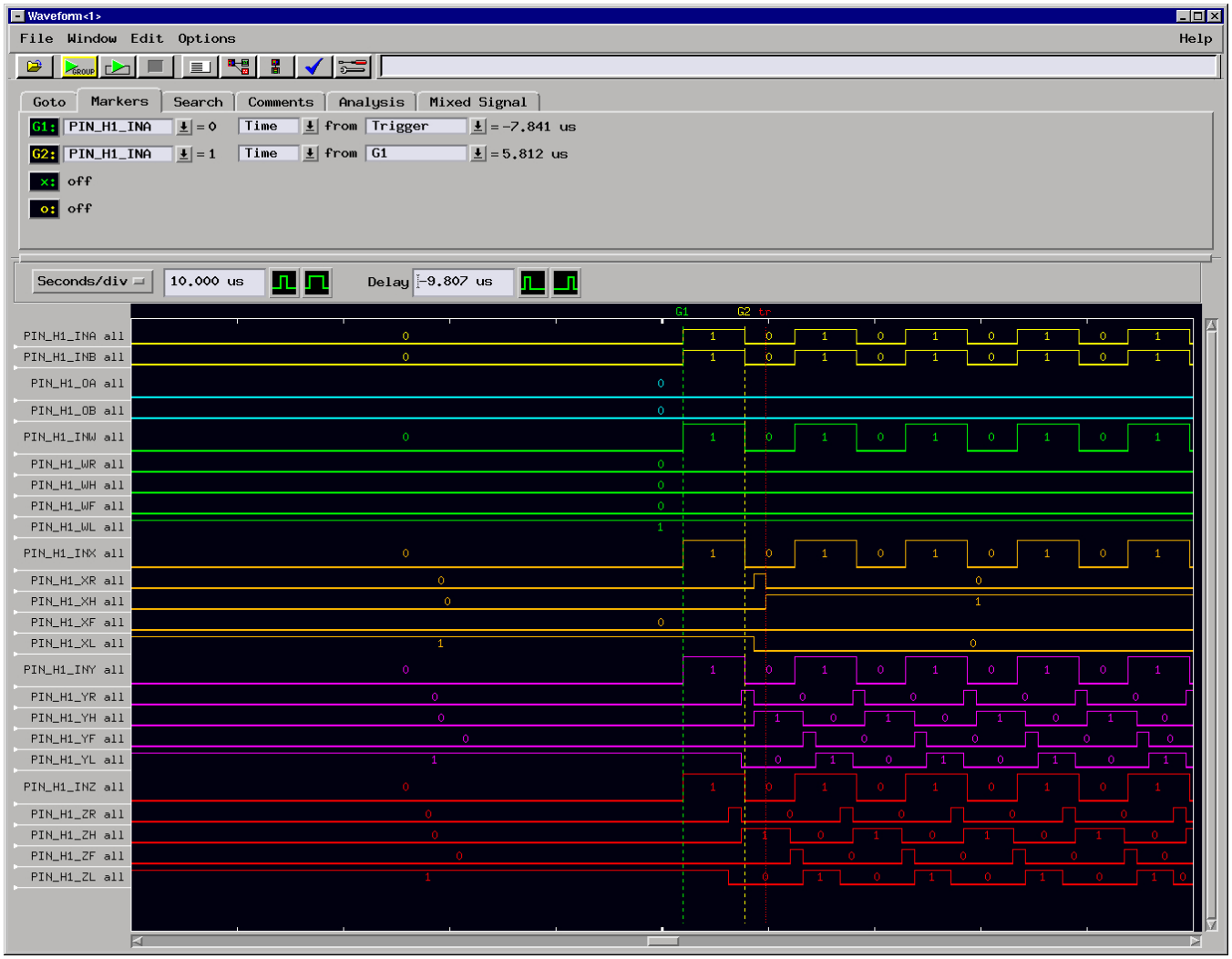


Figure 26. Debounce Test 2 Waveform

8.3 PTO Test Results

The commands issued to the PTO for the three projects ***pto_countdir_app***, ***pto_cwccw_app***, ***pto_quadrature_app*** are listed in Table 7 . This is not an actual motion profile but a series of commands meant to test the PTO.

Table 7. Commands Sequence for PTO Tests

Command #	Initial Step Width	Number of Steps	Direction	Acceleration
1	2000	10	Forward	Linear Deceleration
2	6000	10	Reverse	Linear Acceleration
3	8192	16	Forward	Constant Speed
4	4096	11	Reverse	Linear Deceleration
5	65536	1	Delay Only	Constant Speed
6	4608	10	Forward	Linear Acceleration
7	65536	1	Delay Only	Constant Speed
8	8192	10	Reverse	Linear Acceleration

Figure 27 shows the resulting plot of the measured versus simulated profile for the count/dir mode. Note how the measured and simulated results match exactly and how the acceleration and decelerations are linear. The captured output waveform that was used to make this plot is shown Figure 28. Figure 29 and Figure 30 show the same result in cw/ccw mode. Finally, Figure 31 and Figure 32 show the same result in quadrature mode.

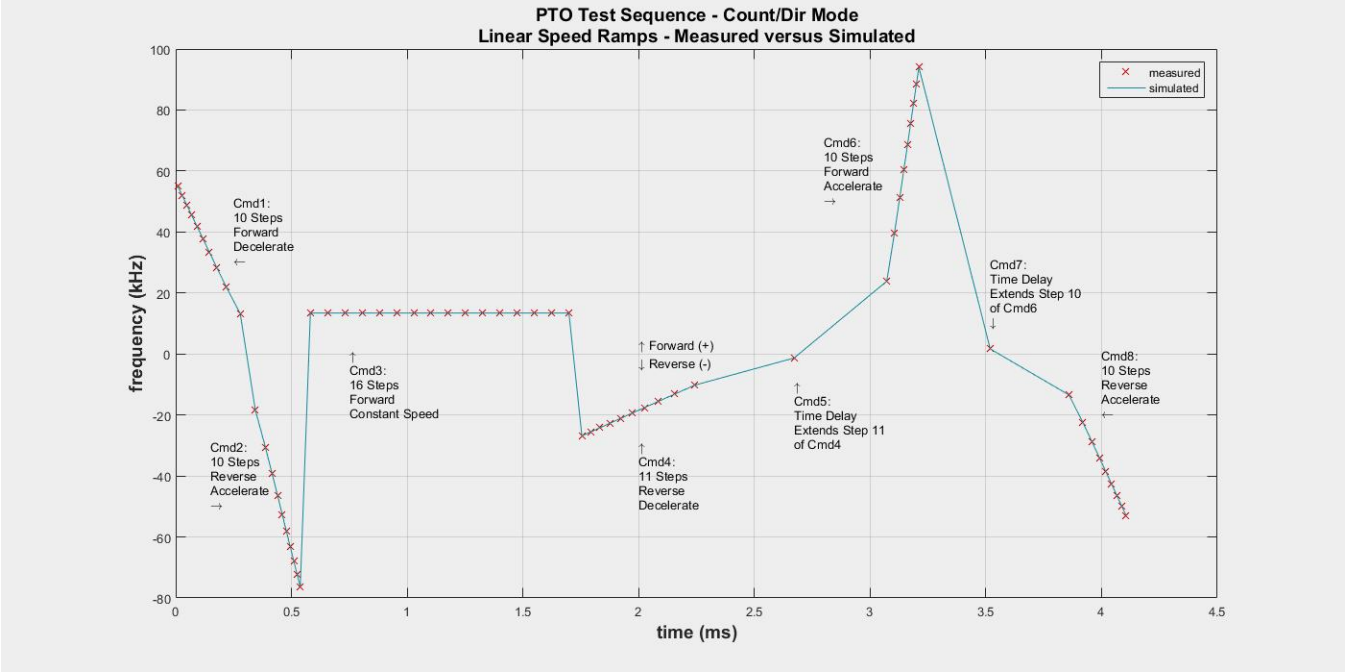


Figure 27. PTO Test Sequence - Measured versus Simulated Data - Count/Dir Mode

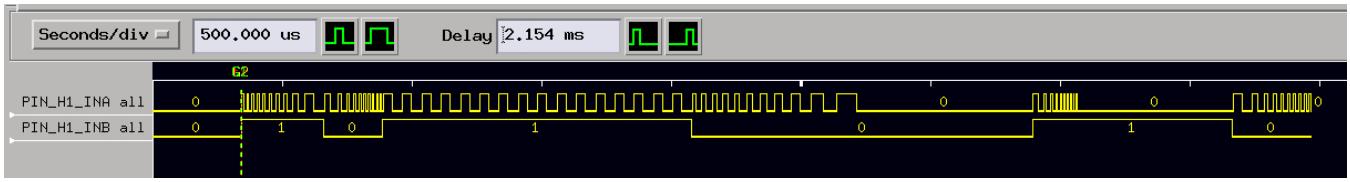


Figure 28. PTO Test Sequence Waveform - Count/Dir Mode

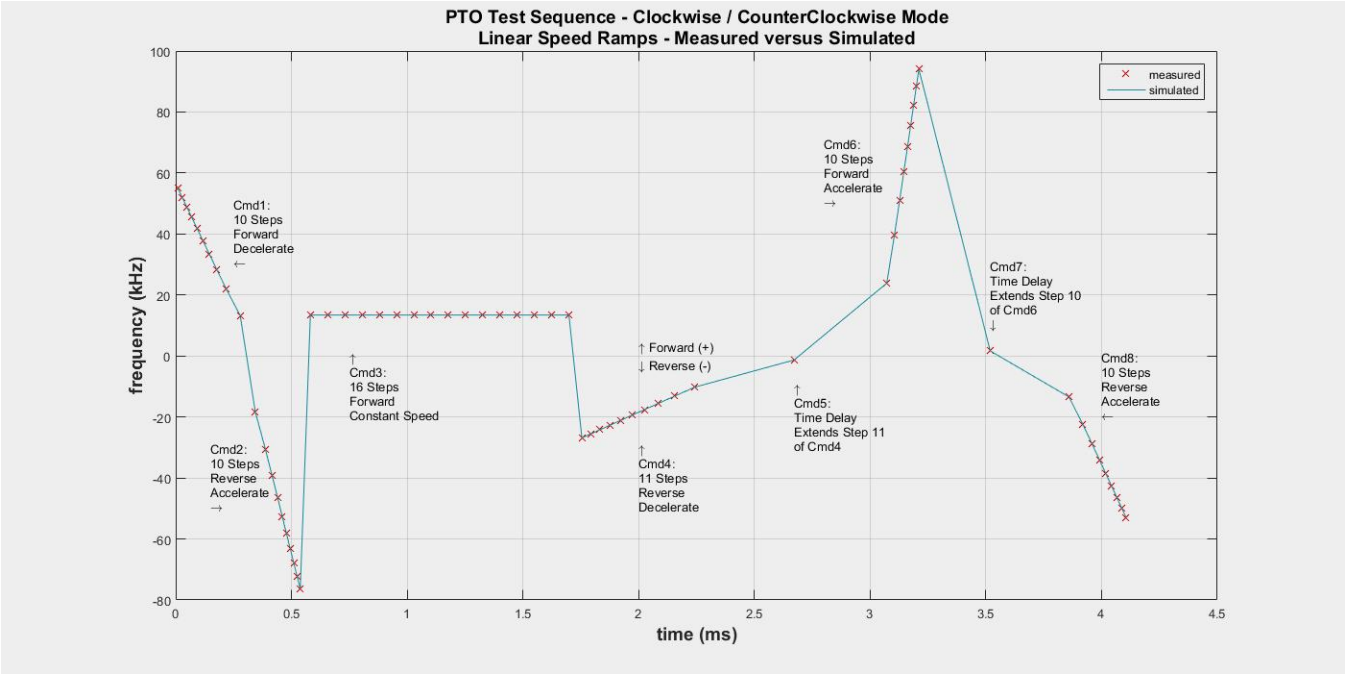


Figure 29. PTO Test Sequence - Measured versus Simulated Data - Cw/Ccw Mode

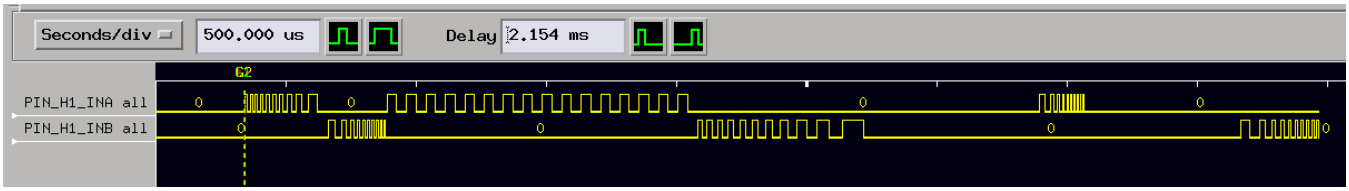


Figure 30. PTO Test Sequence Waveform - Cw/Ccw Mode

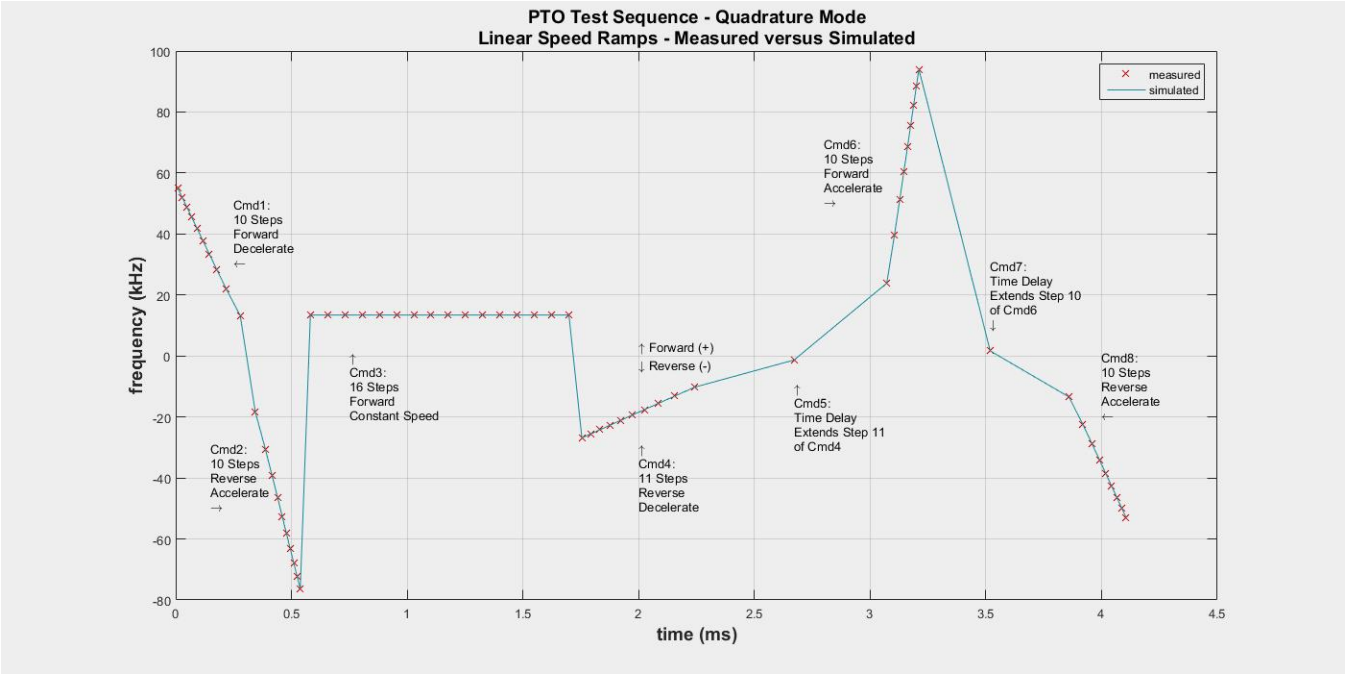


Figure 31. PTO Test Sequence - Measured versus Simulated Data - Quadrature Mode

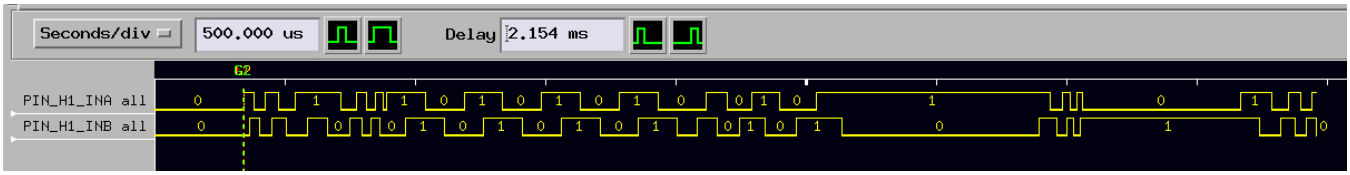


Figure 32. PTO Test Sequence Waveform - Quadrature Mode

9 Design Files

9.1 Schematics

To download the test board schematics in PDF format, see the design files at TIDM-HAHSCPTO.

Schematics for the LaunchPad are available for download at http://processors.wiki.ti.com/index.php/LAUNCHXL2-RM57L.

9.2 Bill of Materials

To download the bill of materials (BOM), see the design files at TIDM-HAHSCPTO. This BOM includes all the components for both the LaunchPad and the test board.

9.3 Layer Prints

To download the test board layer plots, see the design files at TIDM-HAHSCPTO.

9.4 Altium Project

To download the test board Altium project files, see the design files at TIDM-HAHSCPTO.

CAD files for the LaunchPad are available (In EAGLE format) at http://processors.wiki.ti.com/index.php/LAUNCHXL2-RM57L.

9.5 Layout Guidelines

The test board is entirely passive and simply organizes the N2HET signals available on the LAUNCHXL2-RM57L launchpad two booster pack sites and expansion headers into groups of signals suitable for direct connection to logic analyzer and pattern generator pods.

9.6 Gerber Files

To download the Gerber files, see the design files at TIDM-HAHSCPTO.

Gerber files for the LaunchPad are available (In EAGLE format) at http://processors.wiki.ti.com/index.php/LAUNCHXL2-RM57L.

9.7 Assembly Drawings

To download the assembly drawings, see the design files at TIDM-HAHSCPTO

9.8 Software Files

To download the software files, see the design files at TIDM-HAHSCPTO

10 References

1. Embedded.com, *Generate stepper-motor speed profiles in real time*, David Austin, http://www.embedded.com/design/mcus-processors-and-socs/4006438/Generate-stepper-motor-speed-profiles-in-real-time
2. Koren, Israel. *Computer Arithmetic Algorithms*, Brookside Court Publishers, Amherst, Massachusetts, 1998. ISBN 0-13-151952-2
3. Texas Instruments, *High Performance Pulse Train Output (PTO) With PRU-ICSS for Industrial Applications*, Thomas Mauer, Ganesh Mohan Nelliparambil, and Ingolf Frank, www.ti.com/lit/pdf/tidu707
4. Code Composer Studio Download page, http://processors.wiki.ti.com/index.php/Download_CCS
5. HalCoGen, *Hardware Abstraction Layer Code Generator for Hercules MCUs*, Download Page http://www.ti.com/tool/halcogen
6. High End Timer Integrated Development Environment, HET_IDE, Download Page http://www.ti.com/tool/het_ide
7. RM57L843 LaunchPad, http://www.ti.com/tool/LAUNCHXL2-RM57L
8. RM57Lx 16/32 RISC Flash Microcontroller Technical Reference Manual, http://www.ti.com/lit/pdf/spnu562
9. Datasheet, RM57L843 Hercules™ Microcontroller Based on the ARM® Cortex®-R Core
10. MathWorks® MATLAB, http://www.mathworks.com/products/matlab/

11 About the Author

ANTHONY SEELY is an Application Engineer at Texas Instruments, where he is responsible for developing applications for the Hercules family of high-performance microcontrollers. He has over twenty years of experience with Texas Instruments in embedded processing, including automotive microcontrollers and digital signal processors. He earned an MSEE degree from the University of Massachusetts, Amherst and a BSEE degree from the University of Houston.

IMPORTANT NOTICE FOR TI REFERENCE DESIGNS

Texas Instruments Incorporated ('TI") reference designs are solely intended to assist designers ("Designer(s)") who are developing systems that incorporate TI products. TI has not conducted any testing other than that specifically described in the published documentation for a particular reference design.

TI's provision of reference designs and any other technical, applications or design advice, quality characterization, reliability data or other information or services does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such reference designs or other items.

TI reserves the right to make corrections, enhancements, improvements and other changes to its reference designs and other items.

Designer understands and agrees that Designer remains responsible for using its independent analysis, evaluation and judgment in designing Designer's systems and products, and has full and exclusive responsibility to assure the safety of its products and compliance of its products (and of all TI products used in or for such Designer's products) with all applicable regulations, laws and other applicable requirements. Designer represents that, with respect to its applications, it has all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. Designer agrees that prior to using or distributing any systems that include TI products, Designer will thoroughly test such systems and the functionality of such TI products as used in such systems. Designer may not use any TI products in life-critical medical equipment unless authorized officers of the parties have executed a special contract specifically governing such use. Life-critical medical equipment is medical equipment where failure of such equipment would cause serious bodily injury or death (e.g., life support, pacemakers, defibrillators, heart pumps, neurostimulators, and implantables). Such equipment includes, without limitation, all medical devices identified by the U.S. Food and Drug Administration as Class III devices and equivalent classifications outside the U.S.

Designers are authorized to use, copy and modify any individual TI reference design only in connection with the development of end products that include the TI product(s) identified in that reference design. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of the reference design or other items described above may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI REFERENCE DESIGNS AND OTHER ITEMS DESCRIBED ABOVE ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY DESIGNERS AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS AS DESCRIBED IN A TI REFERENCE DESIGN OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TI's standard terms of sale for semiconductor products (http://www.ti.com/sc/docs/stdterms.htm) apply to the sale of packaged integrated circuit products. Additional terms may apply to the use or sale of other types of TI products and services.

Designer will fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of Designer's non-compliance with the terms and provisions of this Notice.

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