Never stop thinking

TriCore

AP32148

Multi-axis motion control

Application Note

V1.1 2012-02

Microcontrollers

Edition 2012-02

Published by
Infineon Technologies AG
81726 Munich, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TC1798

Revision History: V1.1, 2012-02

Previous Version: V1.0

Page	Subjects (major changes since last revision)
	Supports Altium Tasking, Hightec GNU and WindRiver Diab Compiler

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents

1 Preface

This application note describes the implementation of a multi-axis motion controller on the TriCore architecture [1] for the AUDO MAX-family. It explains the configuration of five 3-phase complementary Pulse Width Modulation (PWM) outputs from a single microcontroller. The document is aimed at developers who write or design real-time motion control applications on the TriCore and consider merging the control of multiple axes to one controller. This document looks specifically at those features of the TriCore architecture that make it such an attractive platform for real-time embedded systems, focusing particularly on the peripheral modules: The General Purpose Timer Array (GPTA), the Direct Memory Access (DMA) Module and the Analog to Digital Converter (ADC) but also pointing out the advantages of the CPU core to run the control algorithm.

This guide assumes that readers have access to the TriCore Architecture Manual [2] and the User's Manual [3], and has at least some general knowledge of TriCore instruction set, the architectural features and peripheral modules. The application notes explaining the principles of a single 3-phase PWM setup using the GPTA [3][4] and Field Oriented control [6][7] are particularly pertinent to potential readers of this document.

See References on page 18 for more information on the TriCore and other relevant documentation. It is assumed that most readers will be generally familiar with the features and functions of motion control systems.

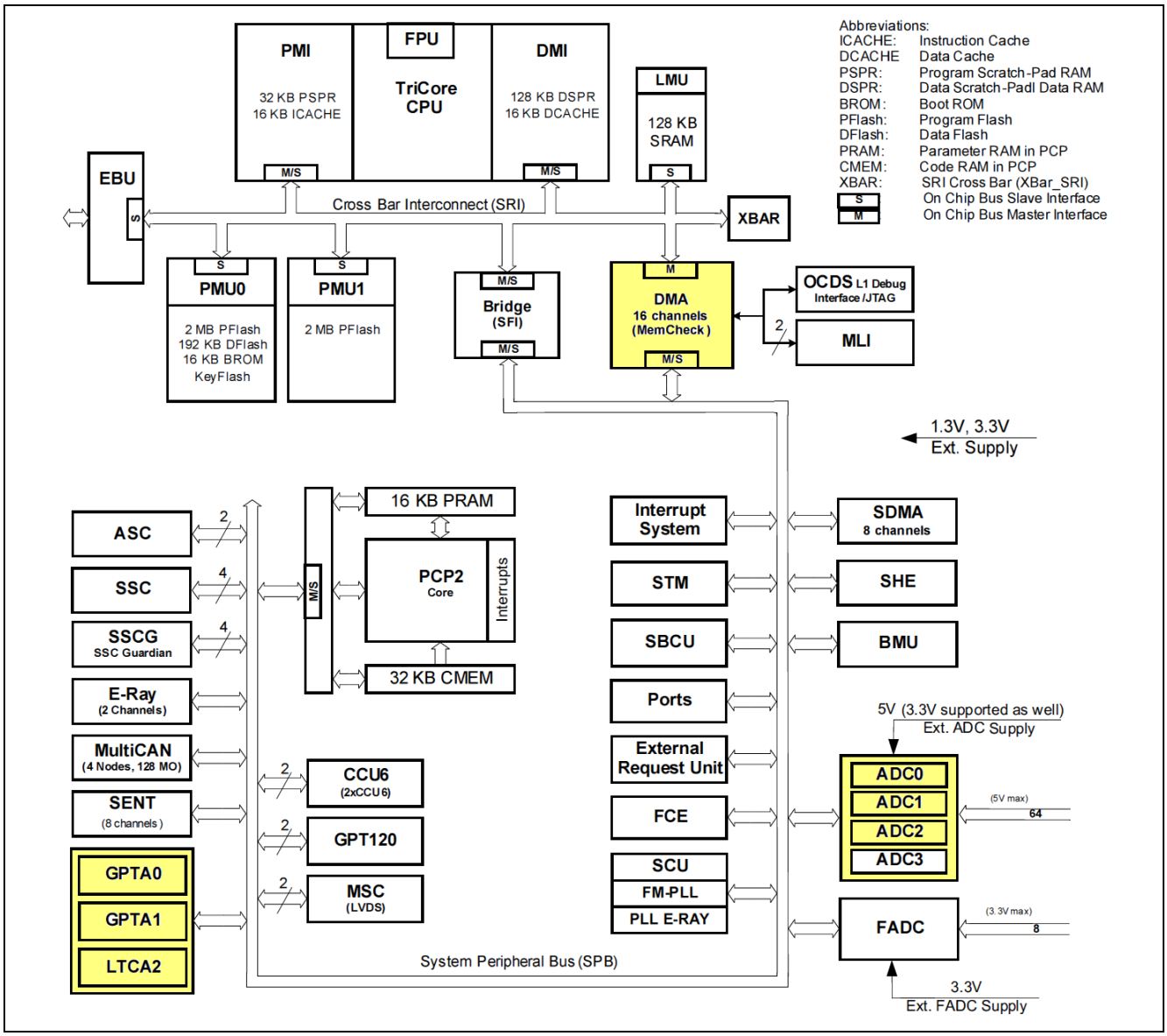

Figure 1 TC1798 Block Diagram

2 Introduction

Figure 1 shows the TC1798 block diagram. Modules used in this application note are marked yellow. This section 2 gives an introduction to the principles of how to generate multiple 3-phase complementary PWM signals on one TriCore. Section 3 explains an efficient configuration and initialization of the GPTA, ADC and DMA module. Section 4 illustrates the example application that is provided with this application note.

Typically, the PWM waveforms drive an H-bridge with high-side and low-side power transistors. To avoid short circuits across this bridge, it is necessary to have a dead time between the complementary waveforms. The three phase currents are measured simultaneously and synchronized to the PWM output.

The GPTA is set up to generate the PWM output and the trigger signal for the ADC. The timing diagram in Figure 2 illustrates five complementary PWM outputs. Only 1 high and low side of each 3-phase PWM is shown. A dead time between the switching on and off of the high and low side switches avoids a short on the power devices. The timer unit also issues a request signal to trigger the ADC. The center of each complementary PWM is shifted by exactly the conversion time of the ADC channel, so that one scan on the master ADC0 over five channels with a synchronized conversion in slave ADC1 and slave ADC2 measures all phase currents in an efficient way. At the end of the last ADC conversion a sequence of transfers in three DMA channels are triggered that moves the ADC results into the TriCore data side memory (LDRAM). The last DMA transfer triggers a TriCore interrupt which executes the control algorithm.

The scan of five ADC conversions at 12-bit resolution requires about 4.9 µs, three DMA transfers about 1 µs and the interrupt including five FOC algorithms about 5.8 µs. The PWM update for the next period is finished 12 µs after the first ADC measurement was triggered. The CPU load is only caused by the control algorithm and reaches less than 10% in total.

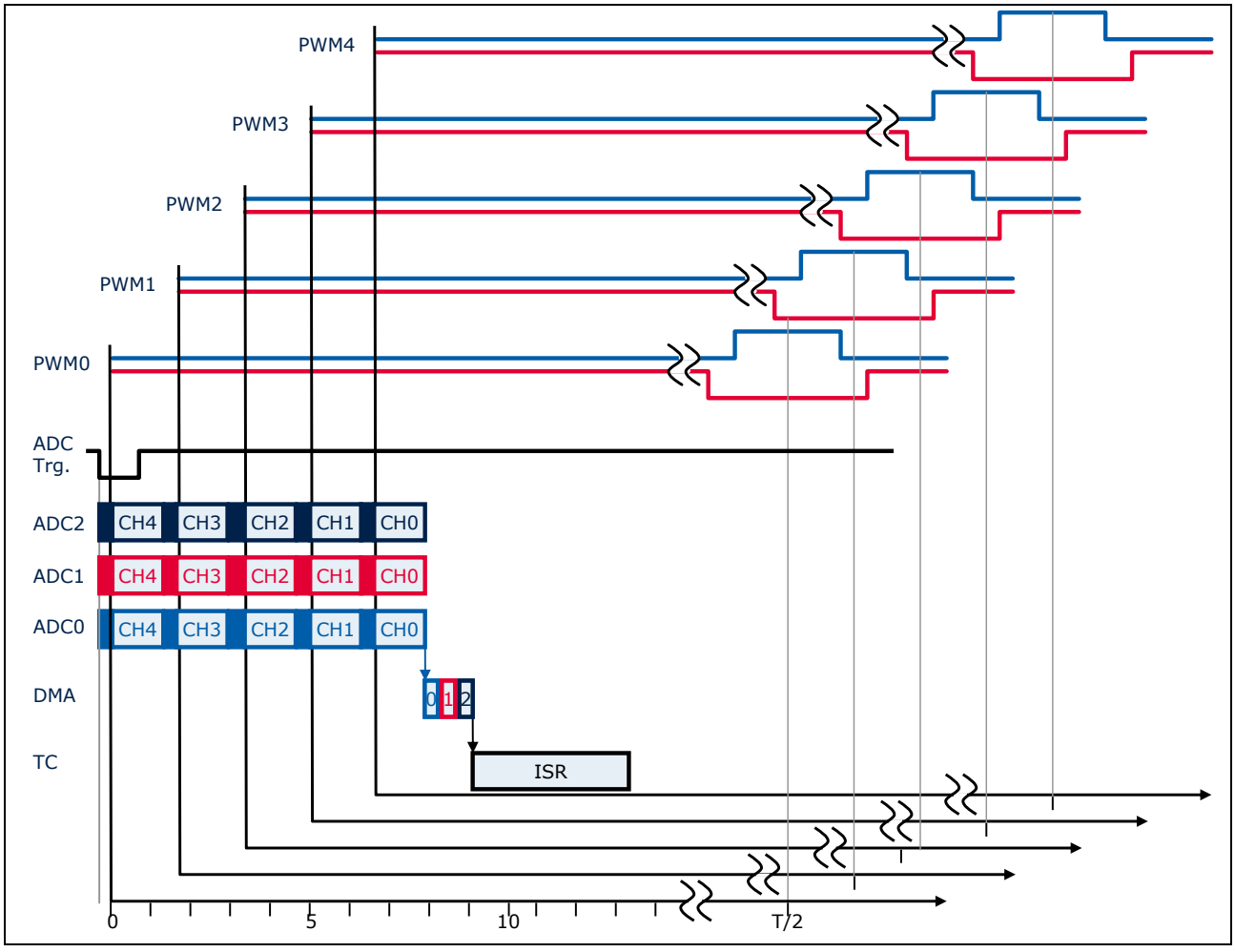

Figure 2 Timing Diagram

3 Configuration

3.1 PWM

The TC1798 contains two General Purpose Timer Arrays (GPTA0 and GPTA1) with identical functionality, plus an additional Local Timer Cell Array (LTCA2). Figure 3 shows a global view of the GPTA modules.

The GPTA provides a set of timer, compare, and capture functionalities that can be flexibly combined to form signal measurement and signal generation units. They are optimized for tasks typical of engine, gearbox, and electrical motor control applications, but can also be used to generate simple and complex signal waveforms required for other industrial applications.

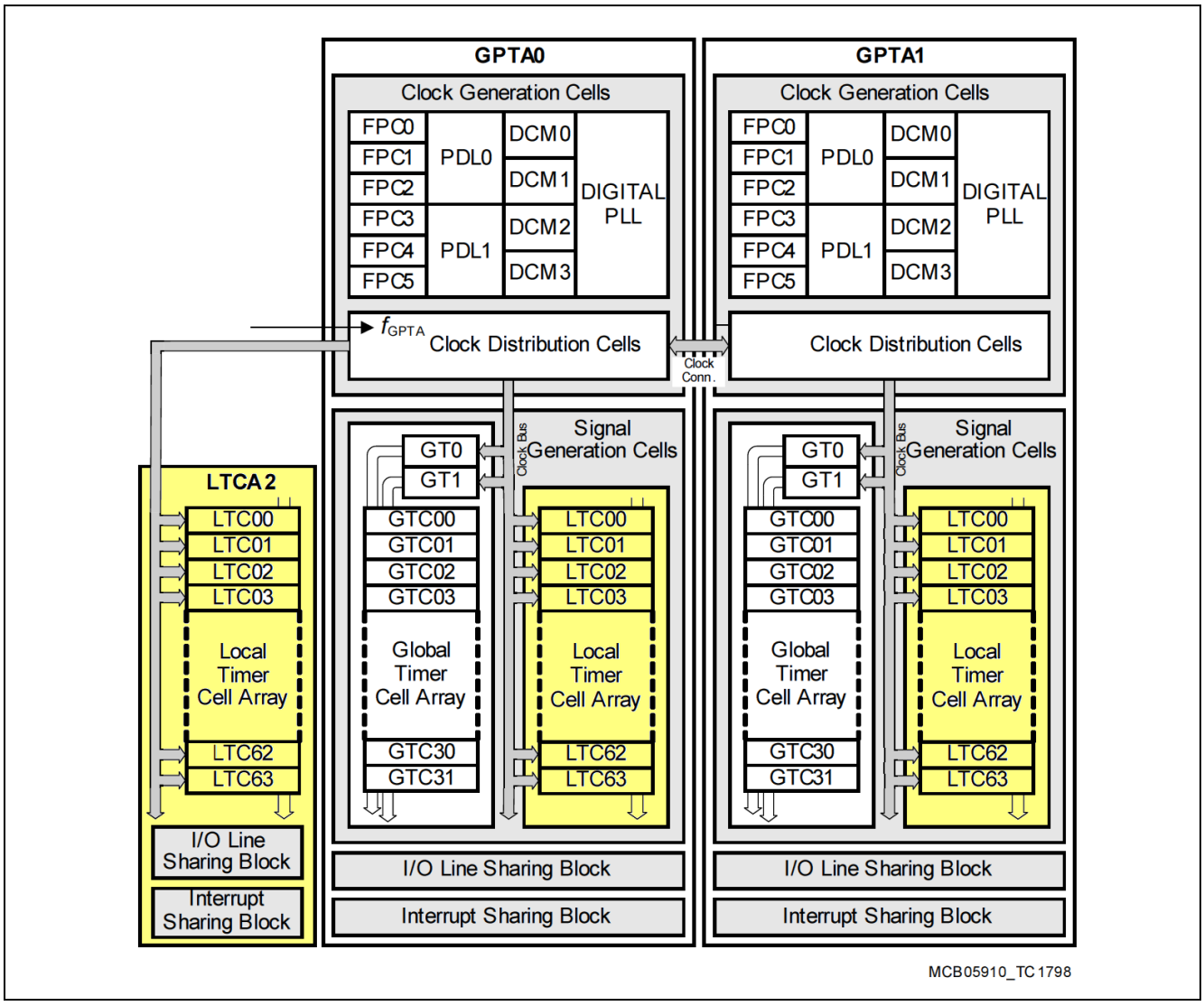

Figure 3 General Block Diagram of the GPTA Modules

Each GPTA is configured to generate two, the LTCA2 one 3-phase complementary PWMs signals. The configuration is shown in Table 1. Each 3-phase PWM requires 26 cascaded LTCs and therefore the GPTA module frequency f_{GPTA} shall be reduced to 45 MHz. The detailed description can be found in [4]. The LTC output is routed through the multiplexer to the ports. In Figure 4 the number of outputs from PWM0 to PMW4 that uses the Output Multiples Groups (OMG) are noted with /n0+n1+n2+n3+n4.

Note: /2,0,2,2,2 for example means 2 signals from PWM0, PWM2, PWM3 and PWM4 are using the OMG11 but none from PWM1.

The ADC trigger signal is generated by LTC53 to LTC56.

Table 1 GPTA0, GPTA1, LTCA2 configuration

				GPTA0			GPTA1			LTCA2		
LTC	**Mode**	**Output Multiplexer Group**	**I/O group**		**Output**	**Port**		**Output**	**Port**		**Output**	**Port**
0	Timer			PWM0			PWM2			PWM4		
1	Period	OMG10	IOG0		(OUT7)	(P2.15)						
2-4	Compare											
5	Compare	OMG10	IOG0		OUT0	P2.8		OUT1	P2.9		OUT2	P2.10
6-8	Compare											
9	Compare	OMG11	IOG1		OUT8	P3.0		OUT9	P3.1		OUT10	P3.2
10-12	Compare											
13	Compare	OMG11	IOG1		OUT11	P3.3		OUT12	P3.4		OUT13	P3.5
14-16	Compare											
17	Compare	OMG12	IOG2		OUT16	P3.8		OUT17	P3.9		OUT18	P3.10
18-20	Compare											
21	Compare	OMG12	IOG2		OUT21	P3.13		OUT22	P3.14		OUT23	P3.15
22-24	Compare											
25	Compare	OMG13	IOG3		OUT24	P4.0		OUT25	P4.1		OUT26	P4.2
26	Timer			PWM1			PWM3					
27	Period											
28-30	Compare											
31	Compare	OMG13	IOG3		OUT30	P4.6		OUT31	P4.7			
32-34	Compare											
35	Compare	OMG24	IOG4		OUT32	P4.8		OUT33	P4.9			
36-38	Compare											
39	Compare	OMG24	IOG4		OUT34	P4.10		OUT35	P4.11			
40-42	Compare											
43	Compare	OMG25	IOG5		OUT40	P8.0		OUT41	P8.1			
44-46	Compare											
47	Compare	OMG25	IOG5		OUT42	P8.2		OUT43	P8.3			
48-50	Compare											
51	Compare	OMG22	IOG2		OUT19	P3.11		OUT20	P3.12			
52	not used											
53	Timer			ADC								
54	Period											
55	Compare											
56	Compare	OMG23	IOG3		OUT28	(P4.4)						

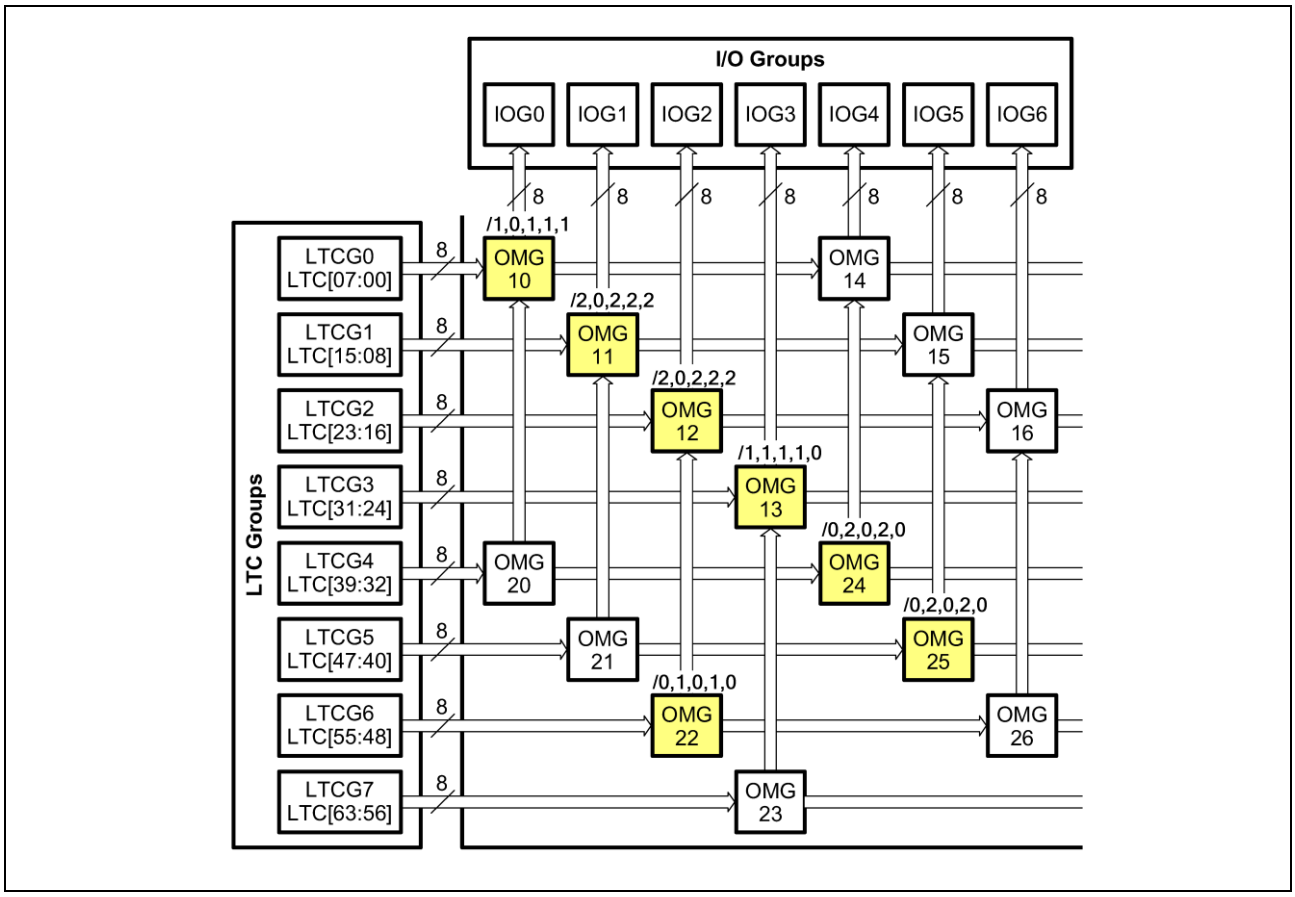

Figure 4 Detail of the Output Multiplexer

The PWM initialization sequence is listed in Listing 1. First the control register of the reset timer is set (Line 106). The associated timer value is initialized in Line 107 with an advanced start time by a multiple of the ADC conversion time value (See also chapter 3.2):

```
#define ADC_CONVERSION_CNTS  49 // (2 * TADC + (4 + STC + n) * TADCI)/TLTC
```

The next local timer cell is initialized as compare cell (Line 109) with a compare value of the PWM period (Line 110).

Listing 2 shows the complete GPTA0 initialization. For module clock configuration see cstart.h. It starts with the PWM0 and PWM1 initialization routine (Line 281,287) and the ADC trigger initialization (Line 289-297). The major part of the listing is the configuration of the output multiplexer. Each byte is related to one GPTA output. 3 bit of the lower nibble determines the 1 out of 8 input lines. 2 bits of the higher nibbles determines the output group: 1 selects one the LTC groups LTCG0 to LTCG3, 2 selects one the LTC groups LTCG4 to LTCG7.

Note: Line 322 for example writes the OMCRL5 value to the FIFO array. It is the FIFO element 35 (see [4] Figure 22-28) which determines OUT40 to OUT43. The OUT42 byte is initialized with 0x27 selecting input 7 of LTCG5, i.e. LTC47.

```
102 void pwm_init(unsigned x, unsigned volatile *ltc_ptr) {
103   unsigned long long *pp;
104   int i;
105
106   *ltc_ptr++ = 0x0103; // Reset Timer, Level sensitive, Toggle SO on overflow
107   *ltc_ptr++ = x * ADC_CONVERSION_CNTS; // Advanced period start
108
109   *ltc_ptr++ = 0x0031; // Compare, SOL/SOH active
110   *ltc_ptr++ = 2 * PWM_PERIOD_CENTER_CNTS;
111
112   pp = (unsigned long long *) ltc_ptr; // double word LTCCTR register distance
113   for (i = 0; i < 3; i++) {
114     // High side
115     *(unsigned*) pp++ = 0x1811; // Compare, SOL active, Set output
116     *(unsigned*) pp++ = 0x3011; // Compare, SOL active, Reset or copy output
117     *(unsigned*) pp++ = 0x3821; // Compare, SOH active, Set or copy output
118     *(unsigned*) pp++ = 0x3021; // Compare, SOH active, Reset or copy output
119     // Low side
120     *(unsigned*) pp++ = 0x1011; // Compare, SOL active, Reset output
121     *(unsigned*) pp++ = 0x3811; // Compare, SOL active, Set or copy output
122     *(unsigned*) pp++ = 0x3021; // Compare, SOH active, Reset or copy output
123     *(unsigned*) pp++ = 0x3821; // Compare, SOH active, Set or copy output
124   }
125 }
```

Listing 1 PWM Configuration

```
276   //
277   // General Purpose Timer Array (GPTA)
278   //
279
280   //  PWM 0
281   pwm_init(0, &GPTA0_LTCCTR00.U);
285
286   //  PWM 1
287   pwm_init(1, &GPTA0_LTCCTR26.U);
288
289   ltc_ptr = &GPTA0_LTCCTR53.U;
290   *ltc_ptr++ = 0x0103;
291   *ltc_ptr++ = 0;
292   *ltc_ptr++ = 0x0131;
293   *ltc_ptr++ = 2 * PWM_PERIOD_CENTER_CNTS;
294   *ltc_ptr++ = 0x1831;
295   *ltc_ptr++ = 2 * PWM_PERIOD_CENTER_CNTS - PRETRIGGER0_CNTS;
296   *ltc_ptr++ = 0x3031;
297   *ltc_ptr = 2 * PWM_PERIOD_CENTER_CNTS - PRETRIGGER1_CNTS;
298
299   GPTA0_MRACTL.B.MAEN = 0; // disable multiplexer array
300   while (GPTA0_MRACTL.B.MAEN != 0)
301     ; // wait for bit MAEN
302   GPTA0_MRACTL.B.WCRES = 1; // reset count
303   GPTA0_MRADIN.U = 0x07000000; // GPTA0_OTMCR1 {0,OUT28_TRIG16,0,0}
304   GPTA0_MRADIN.U = 0; // GPTA0_OTMCR0
305   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH13
306   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL13
307   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH12
308   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL12
309   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH11
310   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL11
311   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH10
312   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL10
313   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH9
314   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL9
315   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH8
```

```
316   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL8
317   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH7
318   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL7
319   GPTA0_MRADIN.U = 0;  // GPTA0_OMCRH6
320   GPTA0_MRADIN.U = 0; // GPTA0_OMCRL6
321   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH5
322   GPTA0_MRADIN.U = 0x00270023; // GPTA0_OMCRL5 {0,LTC47_OUT42,0,LTC43_OUT40}
323   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH4
324   GPTA0_MRADIN.U = 0x00270023; // GPTA0_OMCRL4 {0,LTC39_OUT34,0,LTC35_OUT32}
325   GPTA0_MRADIN.U = 0x00170020; // GPTA0_OMCRH3 {0,LTC31_OUT30,0,LTC56_OUT28}
326   GPTA0_MRADIN.U = 0x00000011; // GPTA0_OMCRL3 {0,0,0,LTC25_OUT24}
327   GPTA0_MRADIN.U = 0x00001500; // GPTA0_OMCRH2 {0,0,LTC21_OUT21,0}
328   GPTA0_MRADIN.U = 0x23000011; // GPTA0_OMCRL2 {LTC51_OUT19,0,0,LTC17_OUT16}
329   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH1
330   GPTA0_MRADIN.U = 0x15000011; // GPTA0_OMCRL1 {LTC13_OUT11,0,0,LTC09_OUT8}
334   GPTA0_MRADIN.U = 0; // GPTA0_OMCRH0
337   GPTA0_MRADIN.U = 0x00000015; // GPTA0_OMCRL0 {0,0,0,LTC05_OUT0}
338   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH7
339   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL7
340   GPTA0_MRADIN.U = 0x0000B000; // GPTA0_LIMCRH6 {0,0,CLK0_LTC53,0}
341   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL6
342   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH5
343   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL5
344   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH4
345   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL4
346   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH3
347   GPTA0_MRADIN.U = 0x00B00000; // GPTA0_LIMCRL3 {0,CLK0_LTC26,0,0}
348   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH2
349   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL2
350   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH1
351   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRL1
352   GPTA0_MRADIN.U = 0; // GPTA0_LIMCRH0
353   GPTA0_MRADIN.U = 0x000000B0; // GPTA0_LIMCRL0 {0,0,0,CLK0_LTC00}
354   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRH3
355   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRL3
356   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRH2
357   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRL2
358   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRH1
359   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRL1
360   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRH0
361   GPTA0_MRADIN.U = 0; // GPTA0_GIMCRL0
362
363   GPTA0_MRACTL.B.MAEN = 1; // enable multiplexer array
```

Listing 2 GPTA0 Initialization

3.2 ADC

Listing 3 shows the complete ADC initialization. For module clock and arbitration mode configuration see initboard.h. The period t_{ARB} of an arbitration round is given by:

t_{ARB} = GLOBCTR.ARBRND × 4 / f_{ADCD}

and configured to

t_{ARB} = 4 / 100 MHZ = 40 ns.

Note: The arbitration of the ADC module in the TC1798 is much faster then in the TC1796. The timing calibration which was required due to the minimum arbitration round of 267ns in the TC1796 described in AP32135 is not necessary.

One input class in each ADC are configured to a 12-bit resolution (Line 209-211)

Five channels in each ADC are configured by the channel control register ADCn_CHCTRx (Line 214-228). Each channel uses the result register with the same number set by bit field RESRSEL and the input class 0 (bit field ICLSEL). ADC0 control registers are set to a synchronization master by the SYNC bit 7. The Synchronization Control Register (SYNCTR) sets ADC0 to a master, ADC1 and ADC2 to slaves (Line 230-232).
The ADC0 is configured for external trigger events (Line 235) from a falling edge on GPTA_TRIG16 (Line 236). The Conversion Request is set-up using a scan request source 1 for channel 4 to 0 (Line 237). The initialization is completed by enabling the arbitration for the request source 1 (Line 238), a wait for the calibration (Line 240-246) which was started in c startup code and a request to the master to switch on the analog part (Line 247). The slaves will be switched on by the master.

Each channel sampling and conversion requires a time of

2 × t_{ADC} + (4 + STC + n) × t_{ADCI} = 0.98µs

with STC = 0, n=12 and t_{ADC} = 1/100 MHz and t_{ADCI} = 6/100 MHz.

The complete scan of five channels therefore is completed after 4.9 µs.

```
204   //
205   // Analog to Digital Converter (ADC)
206   //
207
208   // Input classes
209   ADC0_INPCR0.U = 0x0100; //  12bit input class 0
210   ADC1_INPCR0.U = 0x0100;
211   ADC2_INPCR0.U = 0x0100;
212
213   // Channel configure
214   ADC0_CHCTR0.U = 0x0080; // Use result register 0 , enable synch request
215   ADC0_CHCTR1.U = 0x1080; // Use result register 1 , enable synch request
216   ADC0_CHCTR2.U = 0x2080; // Use result register 2 , enable synch request
217   ADC0_CHCTR3.U = 0x3080; // Use result register 3 , enable synch request
218   ADC0_CHCTR4.U = 0x4080; // Use result register 4 , enable synch request
219   ADC1_CHCTR0.U = 0x0000; // Use result register 0
220   ADC1_CHCTR1.U = 0x1000; // Use result register 1
221   ADC1_CHCTR2.U = 0x2000; // Use result register 2
222   ADC1_CHCTR3.U = 0x3000; // Use result register 3
223   ADC1_CHCTR4.U = 0x4000; // Use result register 4
224   ADC2_CHCTR0.U = 0x0000; // Use result register 0
225   ADC2_CHCTR1.U = 0x1000; // Use result register 1
226   ADC2_CHCTR2.U = 0x2000; // Use result register 2
227   ADC2_CHCTR3.U = 0x3000; // Use result register 3
228   ADC2_CHCTR4.U = 0x4000; // Use result register 4
229
230   ADC0_SYNCTR.U = 0x30; // Evaluate Ready Input R1 and R2. Kernel is a
    synchronization master
231   ADC1_SYNCTR.U = 0x31; // Evaluate Ready Input R1 and R2. Kernel is a
    synchronization slave
232   ADC2_SYNCTR.U = 0x31; // Evaluate Ready Input R1 and R2. Kernel is a
    synchronization slave
```

```
233
234   // Scan
235   ADC0_CRMR1.U = 0xD;// Gate always enabled, ext. trigger, enable interrupt
236   ADC0_RSIR1.U = 0x1200; // Trigger on falling edge of GPTA_TRIG16
237   ADC0_CRCR1.U = 0x1F; // Scan channel 4-0
238   ADC0_ASENR.U = 1 << 1; // Enable Arbitration Slot 1
239   // Wait for Calibration finished
240   while (ADC0_GLOBSTR.B.CAL)
241     ; // wait for calibration finished. Started in cstart.c
242   while (ADC1_GLOBSTR.B.CAL)
243     ; // wait for calibration finished. Started in cstart.c
244   while (ADC2_GLOBSTR.B.CAL)
245     ; // wait for calibration finished. Started in cstart.c
246   // Switch on analog part. Only switch master. Slaves will be switched on by
    master.
247   ADC0_GLOBCTR.U |= 0x00000300; // Analog part switch on
```

Listing 3 ADC Configuration

3.3 DMA

Listing 4 shows the complete DMA initialization. Access ranges are ENDINIT protected and configured in initboard.h. Each of the three DMA channels transfers the five result register of an ADCx. DMA channel 0 is triggered by the service request from the last ADC conversion of the scan. The channel is configured with a 32 Byte circular source buffer starting at ADC0_RESR0 but uses only 5 x 4 Byte. The values are transferred to the destination address in the DMI RAM adc_results[0][0] to adc_results[4][0]. The adc_result array is organized so that the phase currents i_u, i_v, i_w of one motor are located together in the memory so that a single double word access by LD.D can get all three 16-bit values. The DMA channel 0 triggers DMA channel 1 which transfers the results of ADC1 to the adc_results array and triggers a third DMA channel. The third channel transfers the ADC2 results and issues an interrupt on the TriCore.

```
250   //
251   // Direct Memory Access (DMA)
252   //
253
254   DMA_CHCR00.U = 0xD0383005; // 16bit data width, triggered by ADC_SR00,
    source 32Byte circ buffer, destination 16Byte circ buffer
255   DMA_SADR00.U = (unsigned) &ADC0_RESR0.U;
256   DMA_DADR00.U = (unsigned) &adc_results[0][0]; // Channel 00 destination
    address register aligned to 16Byte
257   DMA_ADRCR00.U = 0x65A9; // Source Address Modification Factor
258   DMA_CHICR00.U = 9 << 8 | 2 << 2; // Service to SR09
259
260   DMA_CHCR01.U = 0xD0380005; // 16bit data width, triggered by DMA_SR09
261   DMA_SADR01.U = (unsigned) &ADC1_RESR0.U;
262   DMA_DADR01.U = (unsigned) &adc_results[0][1]; // Channel 00 destination
    address register
263   DMA_ADRCR01.U = 0x65A9; // Source Address Modification Factor
264   DMA_CHICR01.U = 10 << 8 | 2 << 2; // Service to SR10
265
266   DMA_CHCR02.U = 0xD0380005; // 16bit data width, triggered by DMA_SR10
267   DMA_SADR02.U = (unsigned) &ADC2_RESR0.U;
268   DMA_DADR02.U = (unsigned) &adc_results[0][2]; // Channel 00 destination
    address register
269   DMA_ADRCR02.U = 0x65A9; // Source Address Modification Factor
270   DMA_CHICR02.U = 0 << 8 | 2 << 2; // Service to SR00
271
272   DMA_HTREQ.U = 0x7; // DMA Hardware Transaction Request
273   DMA_SRC0.U = 0x00001000 | DMA_INT0; // map 8 channels to 4 nodes. set
    service request control register
```

Listing 4 DMA Configuration

4 Example Application

The example application configures the hardware after reset as described in chapter 3 and enters an endless loop. The motor rotation is simulated by incremented a global angle variable. In a real application this angle value is updated by the position sensor interface. The TriCore interrupt routine implements five basic motor control algorithm (Listing 5) consisting of the load operation of the phase currents from the DMI RAM, an inverse park transformation from the angle modified in the main loop, a space vector modulation and an update of the GPTA LTC registers for the PWM update.

The control algorithm uses an optimized implementation of the Park/Clarke transformation and space vector modulation (Listing 6) described in [5]. The output of the space vector modulation is shown in Figure 5. The signal output is shown in Figure 6.

The execution time for each control algorithm is less than 350 CPU cycles. The total CPU load for the five-axis motion controller less than 10%.

```
139   // PWM0 update
140   i = *(struct i_s*) &adc_results[0][0];
141   p.Angle = rotor_elec_theta[0];
142   ipark_calc(&p);
143   s.Ualpha = p.Alpha;
144   s.Ubeta = p.Beta;
145   svgendq_calc(&s);
149   pwm_update(&GPTA0_LTCCTR00, &GPTA0_LTCXR02.U, s.Ta, s.Tb, s.Tc);
```

Listing 5 Control algorithm and PWM update

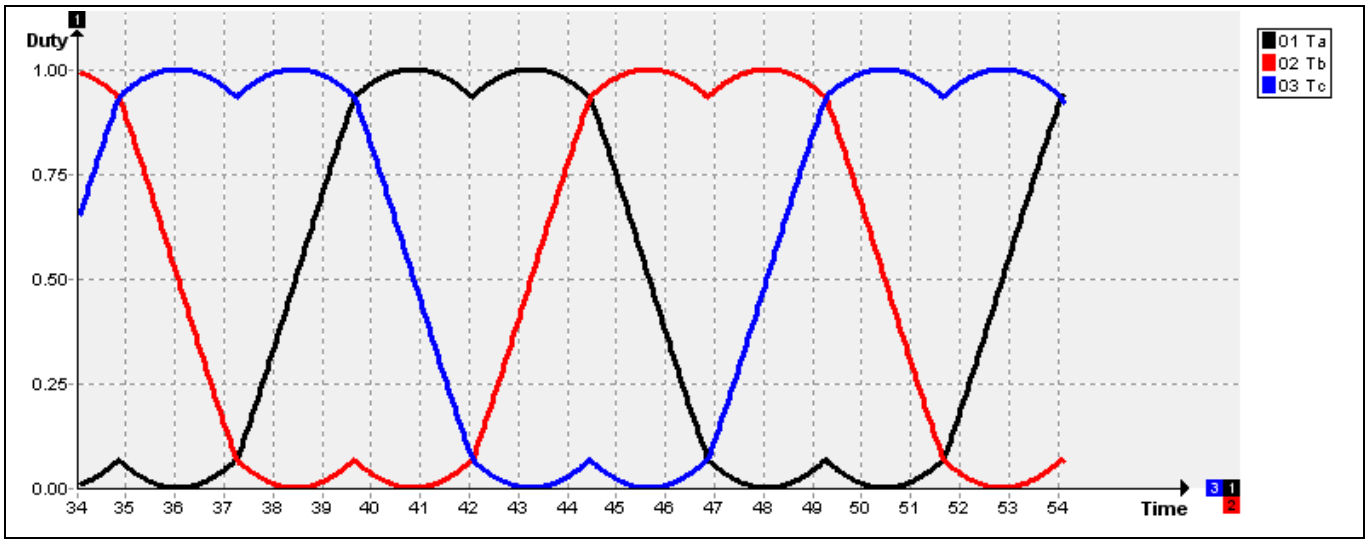

Figure 5 Space vector PWM Modulation

```
 58 void svgendq_calc(SVGENDQ *v) {
 59 #if defined (__TASKING__)
 60     int s;
 61     fract __circ *circ_ptr;
 62     fract t1, t2, t, tt, *p;
 63     fract Ubeta = v->Ubeta;
 64 
 65     Ubeta *= dmc.qseed3;
 66     s = Ubeta >= 0;
 67     s = Ubeta < v->Ualpha ? s + 2 : s;
 68     Ubeta = -Ubeta;
 69     s = v->Ualpha < Ubeta ? s + 4 : s;
 70     s = s ? s : 1;
 71     s = dmc.svgendq_lookup[s-1];
 72 
 73     circ_ptr = __initcirc(v,3*sizeof(fract), s & 0xC);
 74     p = &dmc.svgendq_coeffs[s*2];
 75     t1 = p[0] * v->Ubeta + p[1] * v->Ualpha;
 76     t2 = p[2] * v->Ubeta + p[3] * v->Ualpha + t1;
 77     t = (dmc.svgendq_coeffs[4]-t2)>>1; // t = (1-t2)/2
 78 
 79     s &= 2;
 80     __asm("caddn %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(t),"d"(t1)); // forces
    conditional arithmetic
 81     __asm("cadd %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(tt),"d"(t2)); // forces
    conditional arithmetic
 82     *circ_ptr++ = tt;
 83     __asm("cadd %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(t),"d"(t1)); // forces
    conditional arithmetic
 84     *circ_ptr++ = t;
 85     __asm("caddn %0,%1,%2,%3":"=d"(t):"d"(s),"d"(tt),"d"(t2)); // forces
    conditional arithmetic
 86     *circ ptr = t;
 87 #elif defined(__GNUC__)
 88     int s;
 89     circ t circ ptr;
 90     fract t1, t2, t, tt, *p;
 91     fract Ubeta = v->Ubeta;
 92 
 93     Ubeta =   mulfractfract(Ubeta, dmc.qseed3);
 94     s = Ubeta >= 0;
 95     s = Ubeta < v->Ualpha ? s + 2 : s;
 96     Ubeta = -Ubeta;
 97     s = v->Ualpha < Ubeta ? s + 4 : s;
 98     s = s ? s : 1;
 99     s = dmc.svgendq_lookup[s-1];
100 
101     asm("mov.aa  %L0,%1 \n\
102          mov.a   %H0,%3 \n\
103          addih.a %H0,%H0,%2"
104          :"=a"(circ ptr):"a"(v),"i"(3 * sizeof(long)),"d"(s & 0xC));
105     p = &dmc.svgendq coeffs[s*2];
106     t1 =   adds(  mulfractfract(p[0],v->Ubeta),__mulfractfract(p[1],
         v->Ualpha));
107     t2 =   adds(  adds( _mulfractfract(p[2],v->Ubeta),__mulfractfract(p[3],
         v->Ualpha)),t1);
108     t = (__subs(dmc.svgendq_coeffs[4],t2))>>1;
109 
110     s &= 2;
111       asm("caddn %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(t),"d"(t1)); // forces
    conditional arithmetic
112       asm("cadd %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(tt),"d"(t2)); // forces
    conditional arithmetic
113     circ ptr = put circ long(circ ptr,tt);
114       asm("cadd %0,%1,%2,%3":"=d"(tt):"d"(s),"d"(t),"d"(t1)); // forces
    conditional arithmetic
115     circ ptr = put circ long(circ ptr,t);
116       asm("caddn %0,%1,%2,%3":"=d"(t):"d"(s),"d"(tt),"d"(t2)); // forces
```

```
    conditional arithmetic
117     put circ long(circ_ptr,t);
118 #elif defined(  DCC  )
119 #error diab uses assembler implementation. Include file dmc.s
120 #endif
121     return;
    }
```

Listing 6 Space Vector Modulation – C Source Code (Tasking compiler above. GNUC below)

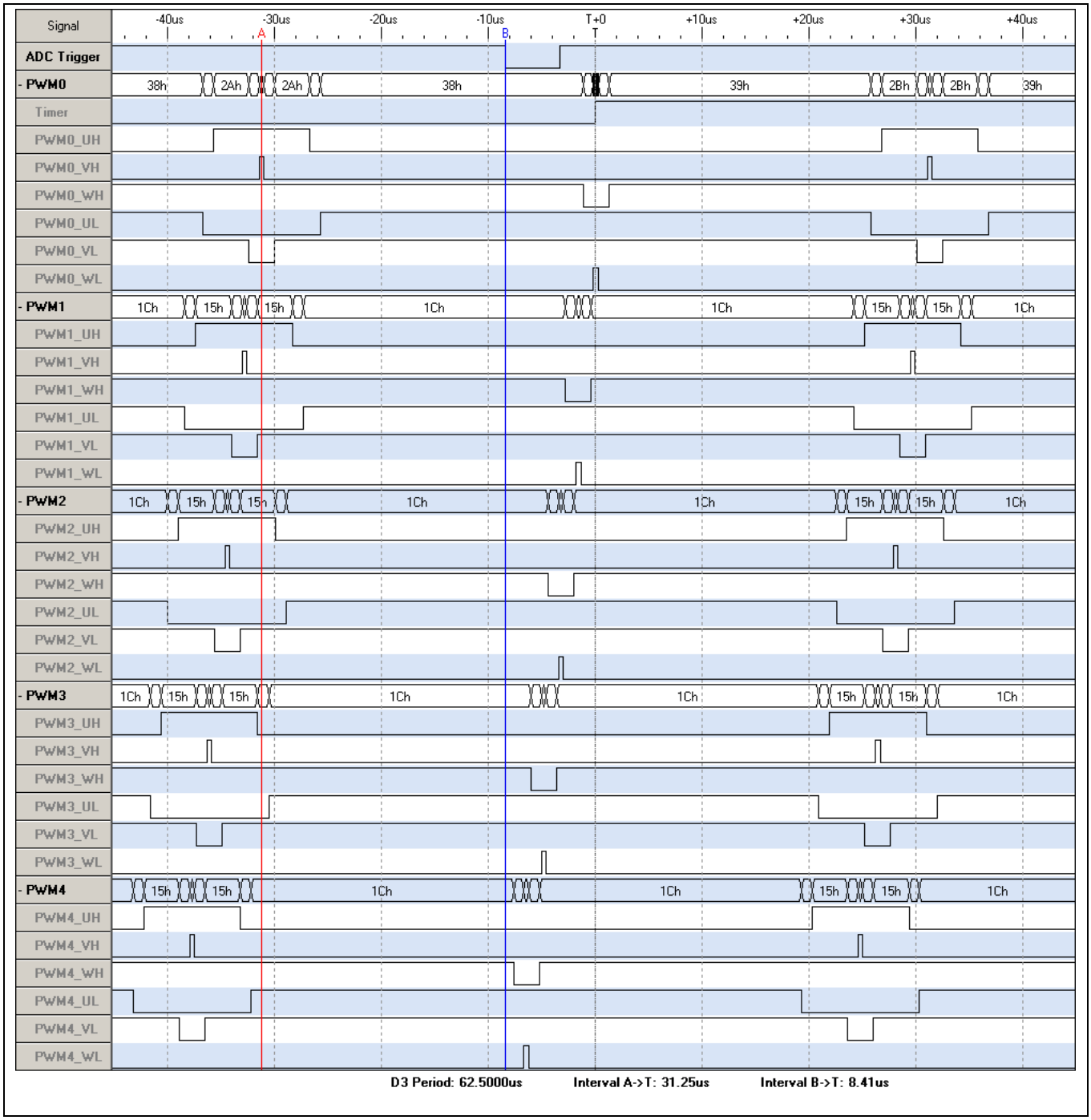

Figure 6 Logic Analyzer showing five 3-phase PWM

5 Tools

The examples were build using the Altium Tasking compiler V4.0, Hightec GNU compiler V4.5 and WindRiver Diab compiler V5.9.1.0. The example code includes project workspaces for the PLS UDE debugger V3.2.

6 Source code

The source code provided with this application note consists of a three makefile based projects for each compiler.

7 References

[1] http://www.infineon.com/tricore

[2] TriCore Architecture V1.3.8 2007-11

[3] Application Note AP32084, TriCore Sinusoidal 3-Phase Output Generation Using the TriCore General Purpose Timer Array

[4] Application Note AP32135, TriCore 3-phase complementary PWM with hardware triggered ADC conversion

[5] TC1797 User's Manual V1.1 2009-05

[6] Application Note AP08059 Sensor less Field Oriented Control for PMSM Motors by using XC886/888,

[7] Application Note AP08090 Sensor less FOC on XC878

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG