The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

AN204827

F²MC-16FX Family, MB96340 Software PWM by use of DMA Transfer

This application note describes the possibility of adding additional, preferably low speed, software PWM channels to the already implemented hardware resources (16bit-PPG). These low speed PWM signals can be for example used for LED or lamp dimming.

Contents

1 Introduction....1
2 Principle....1
3 Software....6
3.1 General....6
3.2 Initialize IO Ports....7
3.3 Initialize Reload Timer....7
3.4 Initialize DMA Channel....8
3.5 Setup PWM Table....10
3.6 Start PWM....10
3.7 Interrupt Handler....10
3.8 Interrupt levels and interrupt table....11
3.9 Main....11
4 Performance....12
4.1 Accuracy....12
4.2 Influence on CPU operation....21
5 Appendix....22
5.1 Additional Information....22
6 Document History....23

1 Introduction

This application note describes the possibility of adding additional, preferably low speed, software PWM channels to the already implemented hardware resources (16bit-PPG). These low speed PWM signals can be for example used for LED or lamp dimming.

By use of a reload timer, a DMA channel and IO ports, up to 16 additional channels can be implemented. Using the DMA function influences CPU operation only very low.

This document explains the basic principle and shows example code for 8/16 channels with 8bit PWM resolution. It is possible, of course, to add more than these 16 channels if more DMA channels and IO ports are used. Also it is possible to change resolution of the PWM signal according to needs and available RAM/ROM space.

Please refer also to the software example 96340_sw_pwm_rlt_dma_io.

The application note and the software example are based on the MB96340 series (MB96F346RSA), but can be easily transferred to other MB96xxx series devices.

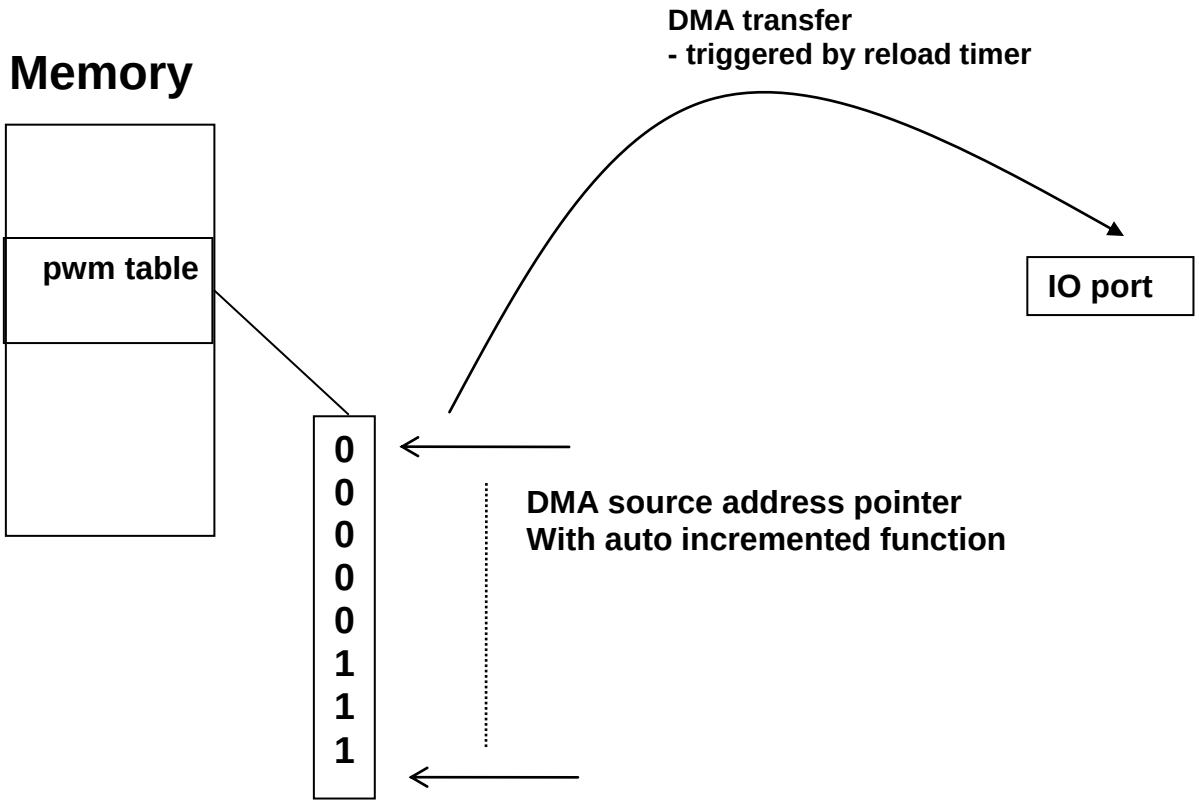
2 Principle

This chapter presents the basic idea of the software PWM.

Basic idea of the software PWM via DMA is the transfer of a predefined PWM table via DMA byte or word transfer from the memory to Port Data Register of one single or two consecutive IO ports of the MCU without CPU interruption.

Depending on application needs the PWM table can be located in Flash ROM (fixed duty value) or in RAM (variable duty value).

Figure 1. Basic Idea

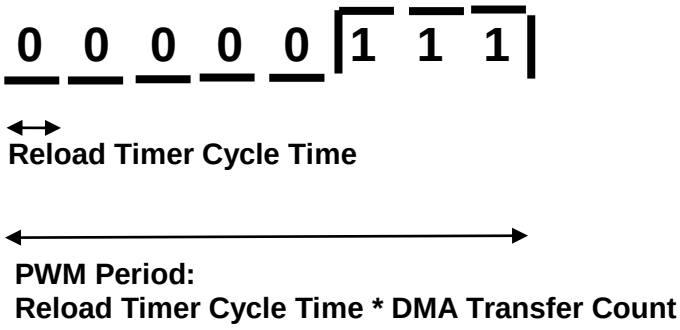


The PWM signal is divided in smaller parts of same length. Number of these parts depends on the resolution. Figure 2 shows an example of 8 parts, which equals 3bit resolution. Most common will be resolution of 8bit which divides the PWM signal into 256 parts.

For each part, the PWM table has to have an entry in the PWM table, which is outputted directly to the IO port by DMA transfer of this value to the Port Data Register of the adequate IO port. This DMA transfer is triggered by an overflow of the reload timer. Therefore the reload timer's cycle time has to be adjusted to the duration of such a timing part.

$$\mathrm{Re}\,loadTimerCycleTime = \frac{PWMFrequency}{2^{RESOLUTION}}$$

Figure 2. PWM Signal



The reload timer overflow generates an interrupt request. This interrupt request is not handled by the CPU, but triggers the automatic DMA transfer of one byte or word. So there is no influence on CPU operation in this case. The reload timer's interrupt request is cleared also automatically by the DMA.

DMA automatically transfers the selected amount of data (byte or word) from the PWM table. DMA source address register has to be automatically updated for each transfer (location in the PWM table), whereas the destination address register keeps on the same value (Port Data Register). DMA transfer count has to be set to $2^{RESOLUTION}$ for byte transfer, 2x $2^{RESOLUTION}$ for word transfer in the beginning. With each DMA transfer, this value is decremented by 1 or 2, depending on transfer width. When the transfer count reaches 0, then the resource interrupt is not handled by the DMA anymore and interrupt request is forwarded to CPU.

Now CPU operation is shortly interrupted to clear reload timer / DMA interrupt request and to reinitialize DMA channel for next PWM cycle. After that, next PWM period is started again via DMA transfer.

Figure 3. PWM Generation

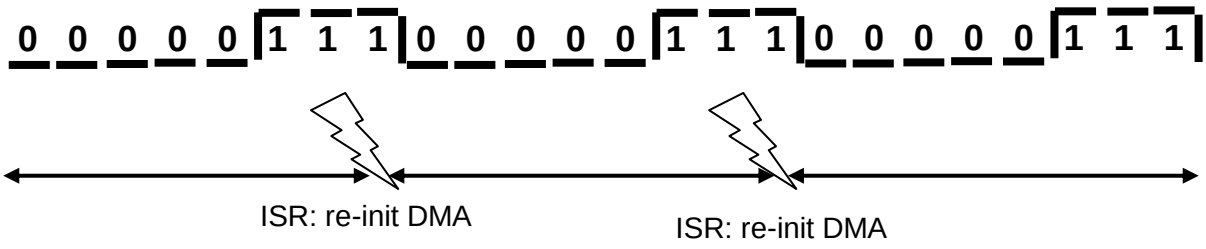


Figure 4. 3bit PWM signal generation

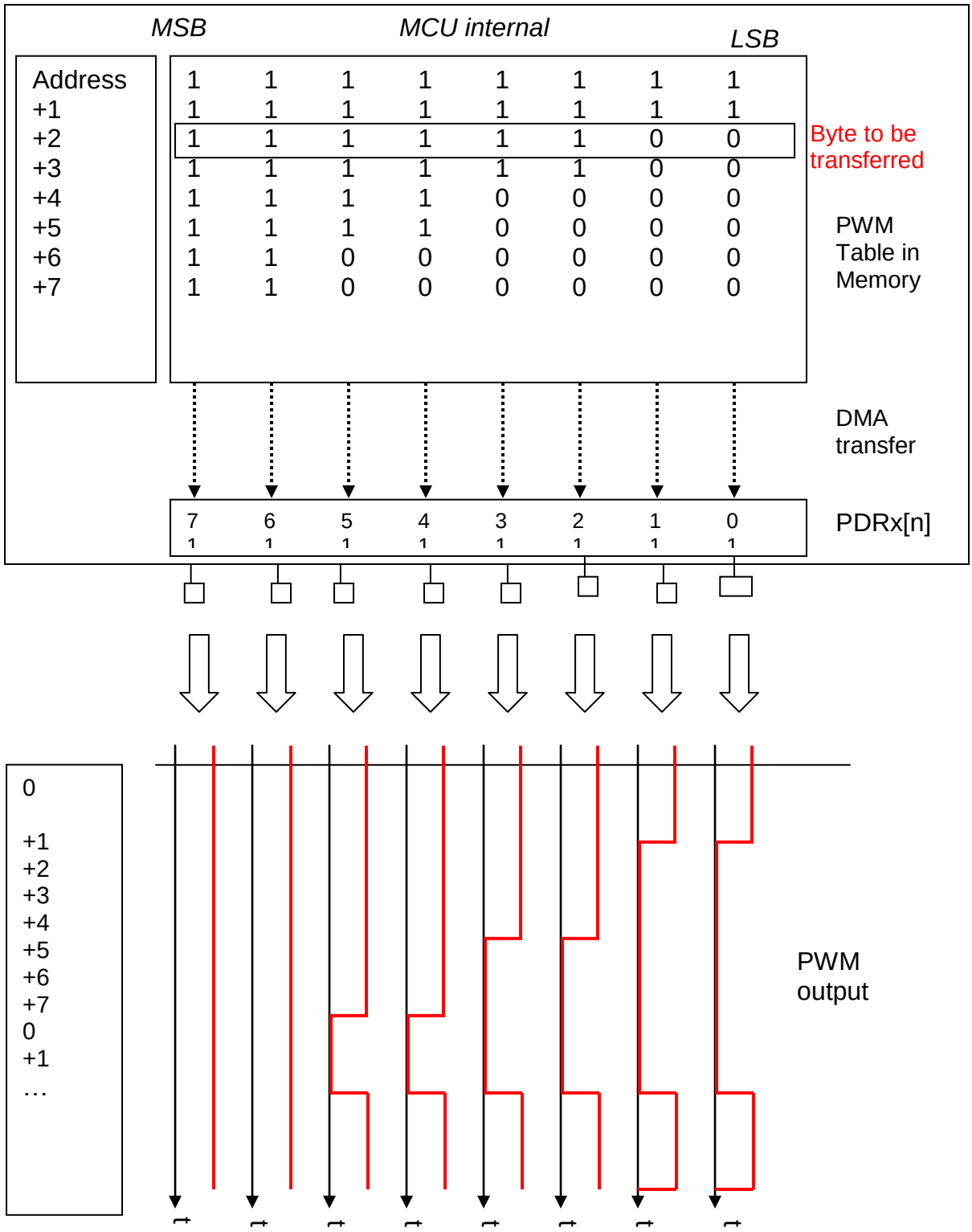


Figure 4 shows the assignment of values in a PWM table in memory to the adequate IO pins for an example with 8 channels and 3bit resolution. With one DMA transfer, one line (equals one byte!) is transferred to the port data register of an IO port and outputted to IO pins.

Figure 5. Extract from SW-PWM signal with 8bit resolution

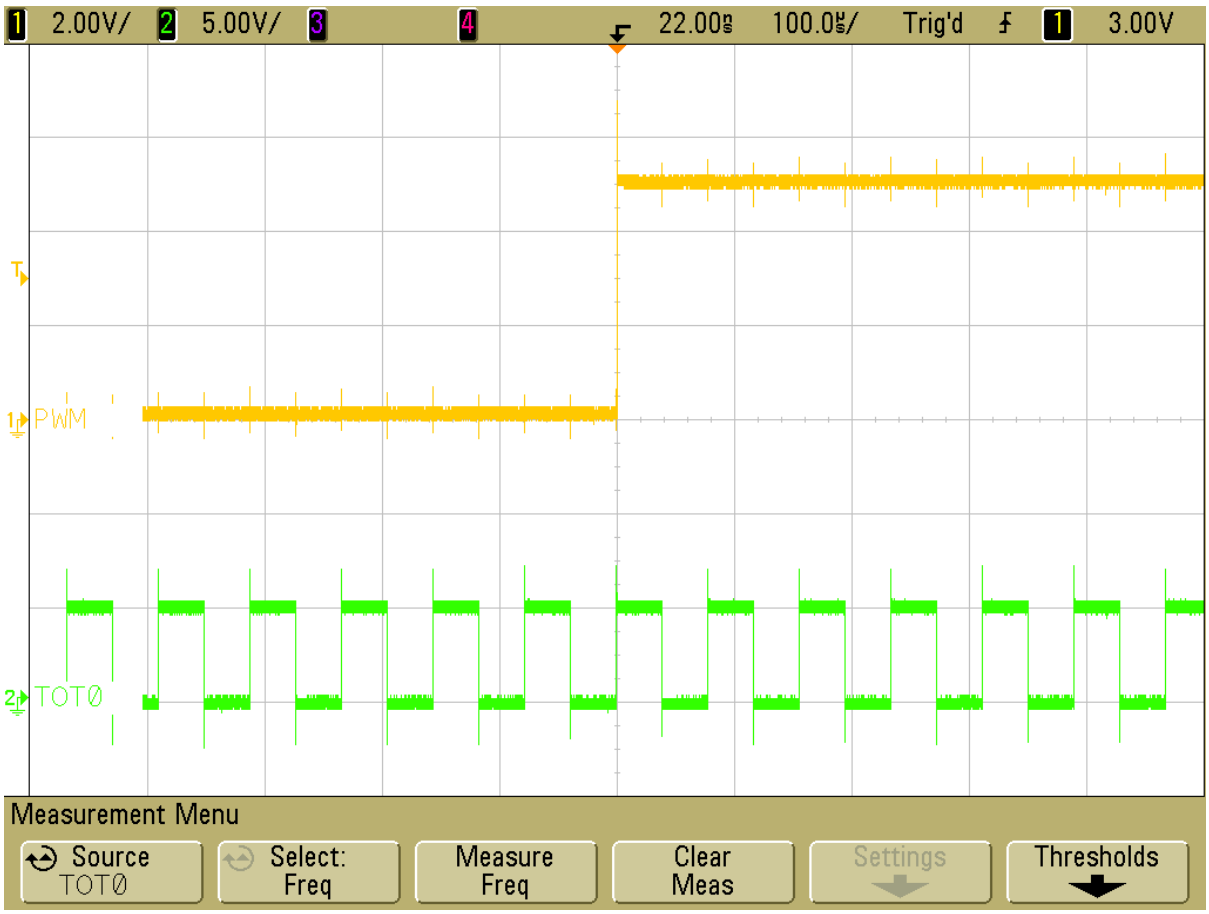


Figure 5 shows a screenshot for an 8bit PWM signal at one single pin. You can see PWM signals rising edge and the output signal of the reload timer. TOT0 pin is toggled with each reload timer interrupt, so each high or low phase equals the reload timer cycle time.

3 Software

This chapter shows example code for realizing software PWM.

3.1 General

For the following sample code, some general definitions are made which include the resolution of the PWM signal, the number of channels and the PWM frequency.

```
/*-----------------------------------------------------------------------*/
/*****   S E T T I N G S  ******                                         */
/*-----------------------------------------------------------------------*/

#define RES8BIT     256u                /* 8bit pwm resolution */
#define RES9BIT     512u                /* 9bit pwm resolution */
#define RES10BIT    1024u               /* 10bit pwm resolution */

/************************************************************************/
#define RESOLUTION  RES8BIT      /* <<**** set pwm resolution here  */
/************************************************************************/

#define CH8         0u           /* 8 pwm channels -> one IO port */
#define CH16  1u           /* 16 pwm channels -> two IO ports */

/************************************************************************/
#define NUMBER_OF_CHANNELS CH8 /* <<*** set number of pwm channels here*/
/************************************************************************/

/************************************************************************/
#define PWM_FREQUENCY      100    /* <<***** set pwm frequency in Hz here  */
                                  /* range from 10Hz to 300Hz will be      */
                                  /* below 0.5% frequency deviation        */
                                  /* with initial values of demo sample    */
/************************************************************************/
```

As stated in the introduction, PWM frequency should be low to minimize impact on CPU operation. Some more information on PWM frequencies and PWM accuracy can be found in chapter 4 of this application note.

The definition of the number of channels will be used later on when initializing the IO ports and setting up the DMA channel.

The value defined as resolution is the number of steps the PWM signal is divided to.

3.2 Initialize IO Ports

The function init_gpio() initializes the used IO pins to output mode. Depending on the selected number of channels one or two consecutive IO ports are used. The ports are set to an initial value and data direction is set to output.

```
void init_gpio(void)
{
       PDR00 = 0x00;              /* set P00-P07 to low level */
       DDR00 = 0xFF;              /* set pins P00-P07 to output */

       #if (NUMBER_OF_CHANNELS == CH16)
       PDR01 = 0x00;       /* set P10-P17 to low level */
       DDR01 = 0xFF;       /* set pins P10-P17 to output */
       #endif
}
```

If you want to use a pin within an IO port assigned to the software PWM in different function, it is possible to use it in resource mode (e.g. analogue input, CAN-TX/RX etc.) or as digital input using the External Pins State Register (EPSRx), not the PDRx register. The adequate bit of the data transferred to the Port Data Register of this port is then not outputted to the pin. It is not possible to use the pin in digital output mode as the DMA transfer regularly would overwrite the PDRxx register with the value defined in the PWM table.

3.3 Initialize Reload Timer

The reload value for the reload timer can be calculated with following formula based on peripheral clock 1 frequency (CLKP1), the PWM frequency and the PWM resolution. It is necessary to select a prescaler value for the input signal of the reload timer.

$$\mathrm{Re}loadValue \approx \frac{CLKP1}{\mathrm{Pr}escaler \cdot 2^{RESOLUTION} \cdot PWMfrequency} - 1$$

Combination of the selected prescaler value and the calculated reload value are the basic values for the PWM accuracy. As reload value has to be rounded to an integer value, for better accuracy try to select a prescaler value that reload value fits best.

For initialization of the reload timer, first stop counter operation. Set reload and prescaler value, set timer to reload mode, clear interrupt flag and enable interrupt request. Set activation by software trigger (done in main function) and enable counter operation.

```
#define PRESCALER2      4u  /* prescaler for reload timer 0: 2^1 = div2 */
#define PRESCALER4      0u  /* prescaler for reload timer 0: 2^2 = div4 */
#define PRESCALER8      5u  /* prescaler for reload timer 0: 2^3 = div8 */
#define PRESCALER16     1u  /* prescaler for reload timer 0: 2^4 = div16 */
#define PRESCALER32     6u  /* prescaler for reload timer 0: 2^5 = div32 */
#define PRESCALER64     2u  /* prescaler for reload timer 0: 2^6 = div64 */

#define RELOAD_PRESCALER PRESCALER4             /* prescaler selection */
#define CLKP1_SPEED         56000000            /* set CLKP1 speed here */

/* set correct divider for reload value calculation */
#if (RELOAD_PRESCALER == PRESCALER2)
        #define DIV_VAL 2lu
#endif
#if (RELOAD_PRESCALER == PRESCALER4)
        #define DIV_VAL     4lu
#endif
#if (RELOAD_PRESCALER == PRESCALER8)
        #define DIV_VAL     8lu
#endif
#if (RELOAD_PRESCALER == PRESCALER16)
        #define DIV_VAL     16lu
#endif
#if (RELOAD_PRESCALER == PRESCALER32)
        #define DIV_VAL     32lu
#endif
#if (RELOAD_PRESCALER == PRESCALER64)
        #define DIV_VAL     64lu
#endif

#define RELOAD_VALUE    CLKP1_SPEED/(DIV_VAL*RESOLUTION*PWM_FREQUENCY)-1
/* calculate reload value for reload timer 0 */

void init_rlt0(void)
{
       TMCSR0_CNTE = 0;      /* stop counter operation */
       TMRLR0 = RELOAD_VALUE;     /* set reload value */
       TMCSR0 = 0x005A | RELOAD_PRESCALER<<10;
       /* set presc., reload mode, interrupt enable, TOT0 output enable */
}
```

3.4 Initialize DMA Channel

Select correct interrupt number as DMA trigger source (51 is reload timer channel 0). Workaround for all unused DMA trigger sources (setting to 12 - delayed interrupt) is only necessary for some 16FX devices.

Set data transfer count to the number of steps ($2^{RESOLUTION}$). Set I/O address pointer to PDR00 register address and buffer address pointer to start address of pwm_table.

Select no I/O pointer update, but buffer pointer update. Set transfer width to byte or word, depending on number of channels. Select direction of transfer from buffer to I/O.

Enable DMA channel operation.

```
/* lookup table for pwm data for 8 channels */

#if (NUMBER_OF_CHANNELS == CH8)
       char pwm_table[RESOLUTION];
#else if (NUMBER_OF_CHANNELS == CH16)
       short int pwm_table[RESOLUTION];
#endif

void init_dma(void)
{
     DER = 0x0000;      /* disable DMA channel 0 */

     DISEL0 = 51;       /* DMA trigger: RLT0 interrupt */

     DISEL1 = 12;       /* set all not used DMA channels to a non-used interrupt
                           source */
     DISEL2 = 12;       /* refer to functional limitation list for details */
     DISEL3 = 12;       /* this workaround is not needed for all devices */
     DISEL4 = 12;
     DISEL5 = 12;

     DCT0 = RESOLUTION;           /* transfer count 256/512/1024 bytes for
                                     8/9/10bit resolution */

     IOA0 = (unsigned int) &PDR00;/* IO address: Port0 data register */

     /* buffer address: pwm lookup table */
     BAPH0 = (__far unsigned long) &pwm_table[0] >> 16;
     BAPM0 = (__far unsigned long) &pwm_table[0] >> 8;
     BAPL0 = (__far unsigned long) &pwm_table[0] & 0xFF;

     DMACS0 = 0x12 | (NUMBER_OF_CHANNELS<<3);
     /* no IOA update, BAP update, transfer width, BAP -> IOA */

     DER = 0x0001;                /* enable DMA channel 0 */
}
```

3.5 Setup PWM Table

Initialize the pwm_table with 0xFF, which equals 100% duty cycle. On SK-16FX-100PMC this equals all LEDs on 7-segment display off.

```
int pwm_config[(NUMBER_OF_CHANNELS+1)*8];
/* save actual configuration of pwm channel */

void init_pwm_table(void)
{
    volatile int j;

    for(j=0;j<RESOLUTION;j++)
    {
        #if (NUMBER_OF_CHANNELS == CH8)
               pwm_table[j] = 0xFF;
               /* fill table with 1 -> all LEDs off */
        #else if (NUMBER_OF_CHANNELS == CH16)
               pwm_table[j] = 0xFFFF;
               /* fill table with 1 -> all LEDs off */
        #endif
    }

     for(j=0;j<(NUMBER_OF_CHANNELS+1)*8;j++)
     {
        pwm_config[j] = RESOLUTION;       /* pwm duty cycle: 100% */
     }
}
```

3.6 Start PWM

Function start_pwm() triggers the reload counter.

```
void start_pwm (void)
{
     TMCSR0_TRG = 1;              /* start RLT0 */
}
```

3.7 Interrupt Handler

In the interrupt handler routine the DMA channel is re-initialized for next transfer. Clear first reload timer interrupt flag and then clear DMA interrupt.

```
__interrupt void irq_rlt_dma (void)
{
     init_dma();             /* re-init DMA channel */
     TMCSR0_UF = 0;          /* Clear reload timer 0 interrupt request */
     DSR = 0x0000;           /* Clear DMA interrupt */
}
```

3.8 Interrupt levels and interrupt table

Set interrupt level for reload timer 0 to a value below 7. Set correct interrupt vector for reload timer 0 in interrupt vector table.

```
#define MIN_ICR  12
#define MAX_ICR  96

#define DEFAULT_ILM_MASK 7

void InitIrqLevels(void)
{
       volatile int irq;
       for (irq = MIN_ICR; irq <= MAX_ICR; irq++)
       {
       ICR = (irq << 8) | DEFAULT_ILM_MASK;
       }
       ICR = 51<<8 | 6;     /* change interrupt level of reload timer 0 */
}

__interrupt void DefaultIRQHandler (void);
__interrupt void irq_rlt_dma (void);

...
#pragma intvect DefaultIRQHandler 50   /* PPG15                          */
#pragma intvect irq_rlt_dma       51   /* RLT0                           */
#pragma intvect DefaultIRQHandler 52   /* RLT1                           */
...
```

3.9 Main

Function main() calls all the initialization functions, globally enables the interrupt an then triggers the reload timer. Then it runs in a while loop, which can be replaced by other code to be executed.

```
void main(void)
{
    InitIrqLevels();
    __set_il(7);           /* allow all levels           */

    init_gpio();           /* initialize IO ports */
    init_rlt0();           /* initialize Reload Timer 0 */
    init_dma();            /* initialize DMA channel 0 */
    init_pwm_table();      /* initialize PWM lookup table-duty cycle 100% */

    __EI();                /* globally enable interrupts */

    start_pwm();           /* start pwm generation */

    while(1)
    {
                           /* here can be your source code */
    }
}
```

4 Performance

This chapter gives information on accuracy and performance influence.

4.1 Accuracy

4.1.1 Accuracy given by prescaler and reload value

As the reload timer is the time base for the software PWM, it is necessary to select prescaler and reload value so that a minimum difference between calculated and real timing is reached.

Example 1: CLKP1 = 56MHz, prescaler = 4; PWM = 100Hz, 8bit resolution, automatic calculation like shown in the example before (take only integer part of result)

$$\mathrm{Re}\,loadValue \approx \frac{56000000}{4 \cdot 2^8 \cdot 100} - 1 = 545.875 \approx 545$$

➔ $f_{RLT} = \frac{56000000}{4 \cdot \mathrm{Re}\,loadValue} = 25688Hz \rightarrow 12844Hz@TOT0$

➔ $f_{PWM} = \frac{56000000}{4 \cdot 2^8 \cdot \mathrm{Re}\,loadValue} = 100.34Hz$

Reload timer toggles TOT0 pin once each timer period by reaching an overflow and generating an interrupt. So only half the frequency of the timer can be seen at the pin!

Example 2: CLKP1 = 56MHz, prescaler = 4; PWM = 100Hz, 8bit resolution, manual calculation (round up to nearest integer), duty value set to 50%

$$\mathrm{Re}\,loadValue \approx \frac{56000000}{4 \cdot 2^8 \cdot 100} - 1 = 545.875 \approx 546$$

➔ $f_{RLT} = \frac{56000000}{4 \cdot \mathrm{Re}\,loadValue} = 25641Hz \rightarrow 12882Hz@TOT0$

➔ $f_{PWM} = \frac{56000000}{4 \cdot 2^8 \cdot \mathrm{Re}\,loadValue} = 100.16Hz$

If the reload value is closer to an integer value, the reload timer cycle time and therefore the PWM signal frequency itself will fit to the expected values.

Figure 6 and Figure 7 show screenshots for the settings from example 2. As you can see, the reload timer output and the PWM signal very well fit to expected values.

Figure 6. Reload Timer output for example 2

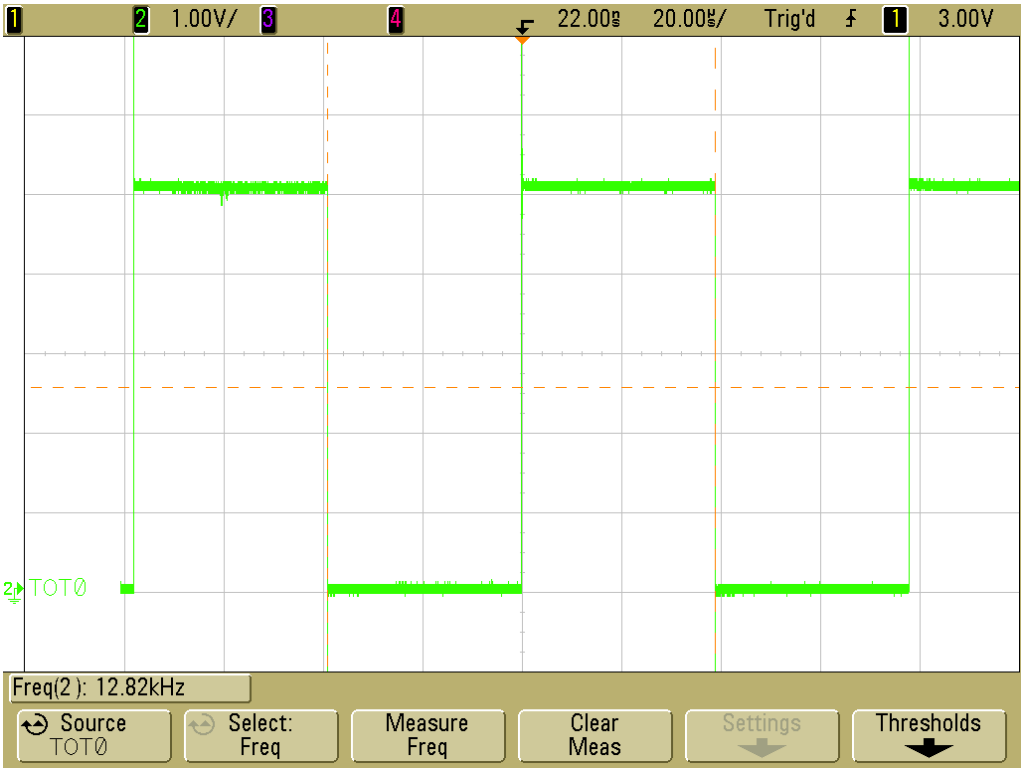


Figure 7. PWM signal output at IO pin for example 2

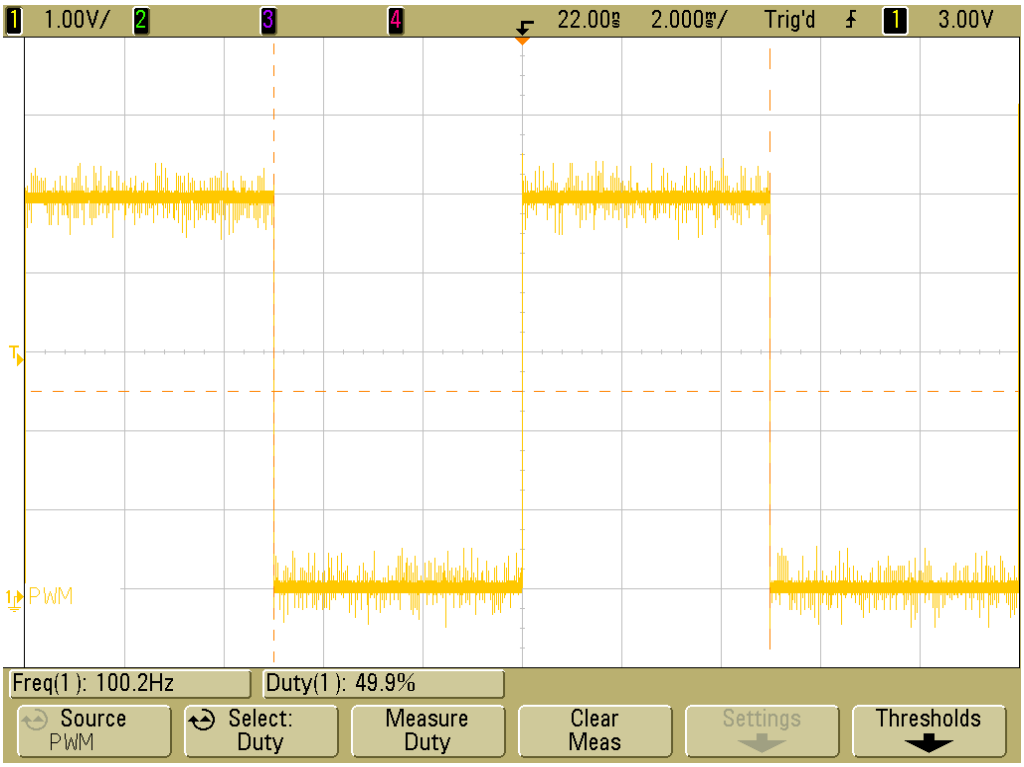


Figure 8 to Figure 11 show more detailed screenshots of the rising and falling edges of the PWM signal (triggered on PWM signal). As you can see, there is a delay between the edge of reload timer output signal and edges of the PWM signal.

At rising and falling edge of PWM signal, there is a fixed delay of about 140ns (176ns-36ns @ rising edge, 195ns-54ns @ falling edge). These 140ns, which are around 8 internal clock cycles (CLKB & CLKP1; 17ns @ 56MHz) are needed internally for detecting the interrupt signal, triggering the DMA transfer, reading data from RAM and transferring to PDR register to output the changed signal.

You can see also a jitter in the delay of +36ns for the rising edge (~2 CPU cycles) and +54ns for the falling edge (~3 CPU cycles), which sum up to a maximum jitter of +90ns for one period of the PWM signal. Taking the long period time of 10ms @ 100Hz as well as taking the resolution of ~78ms (reload timer cycle time) into account, this small jitter can be disregarded in nearly all cases.

Figure 8. Rising Edge for example 2 – Jitter

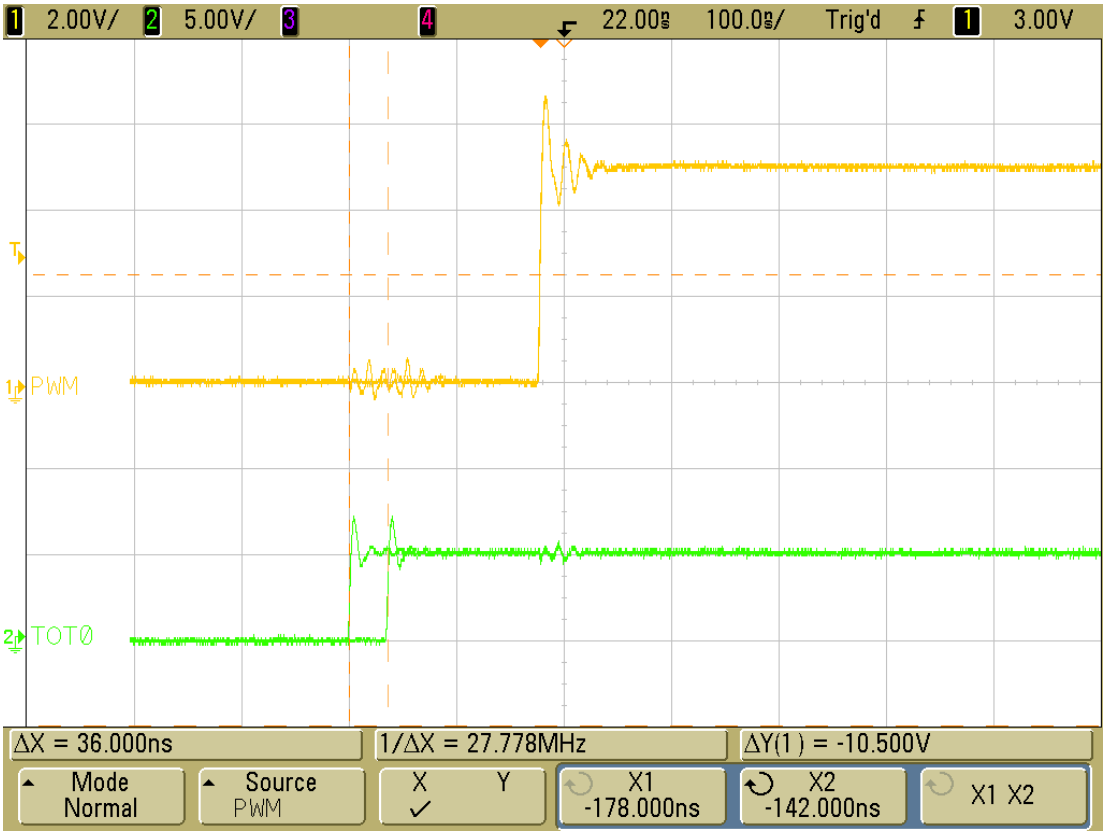


Figure 9. Rising Edge for example 2 – Delay

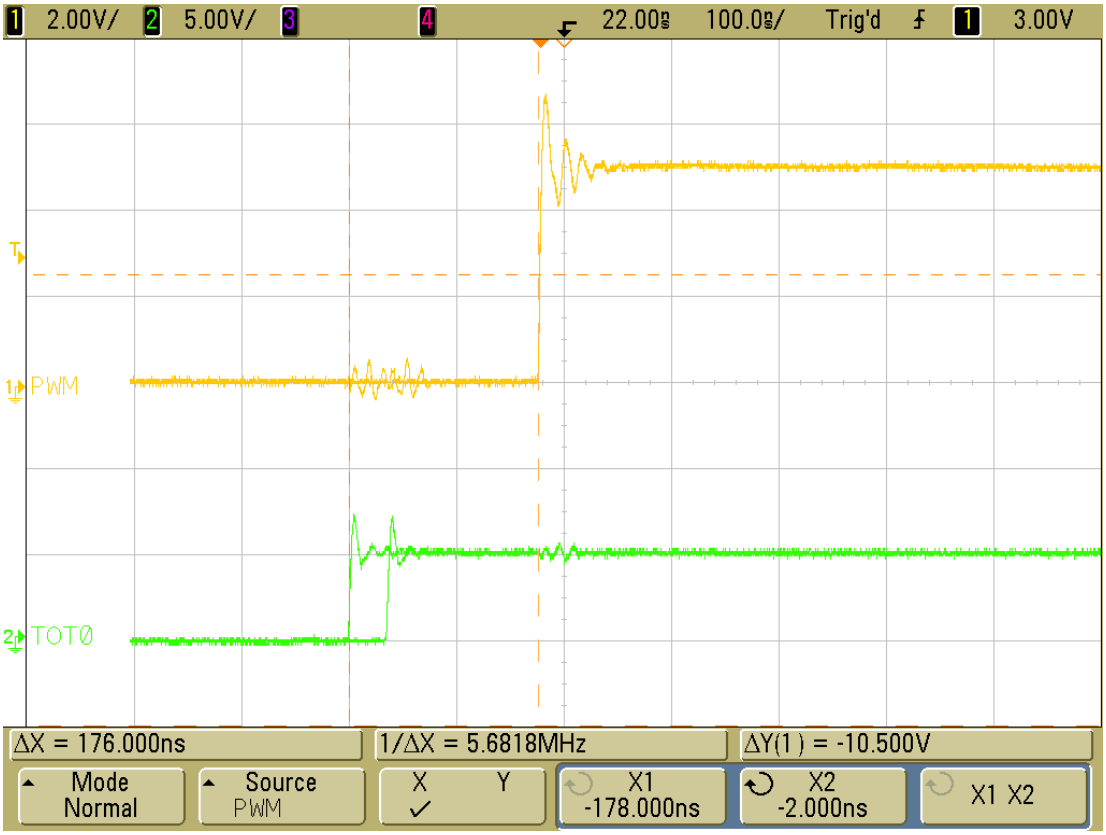


Figure 10. Falling Edge for example 2 - Jitter

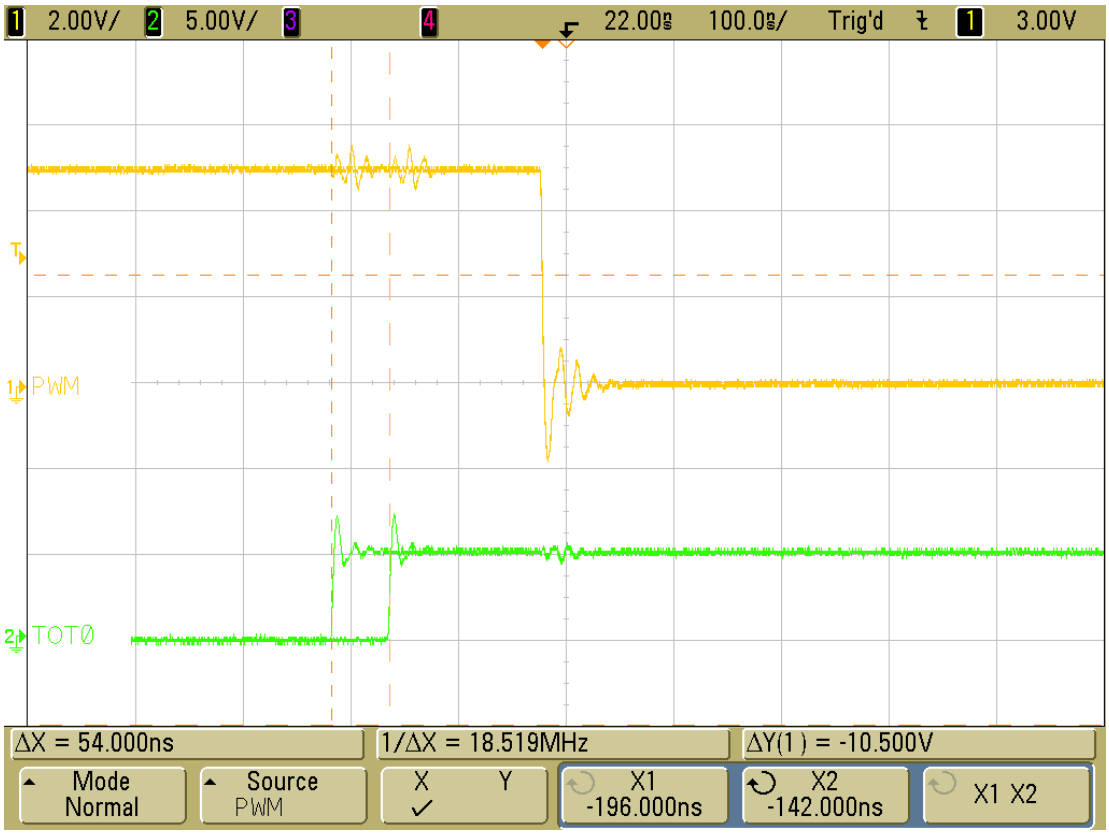
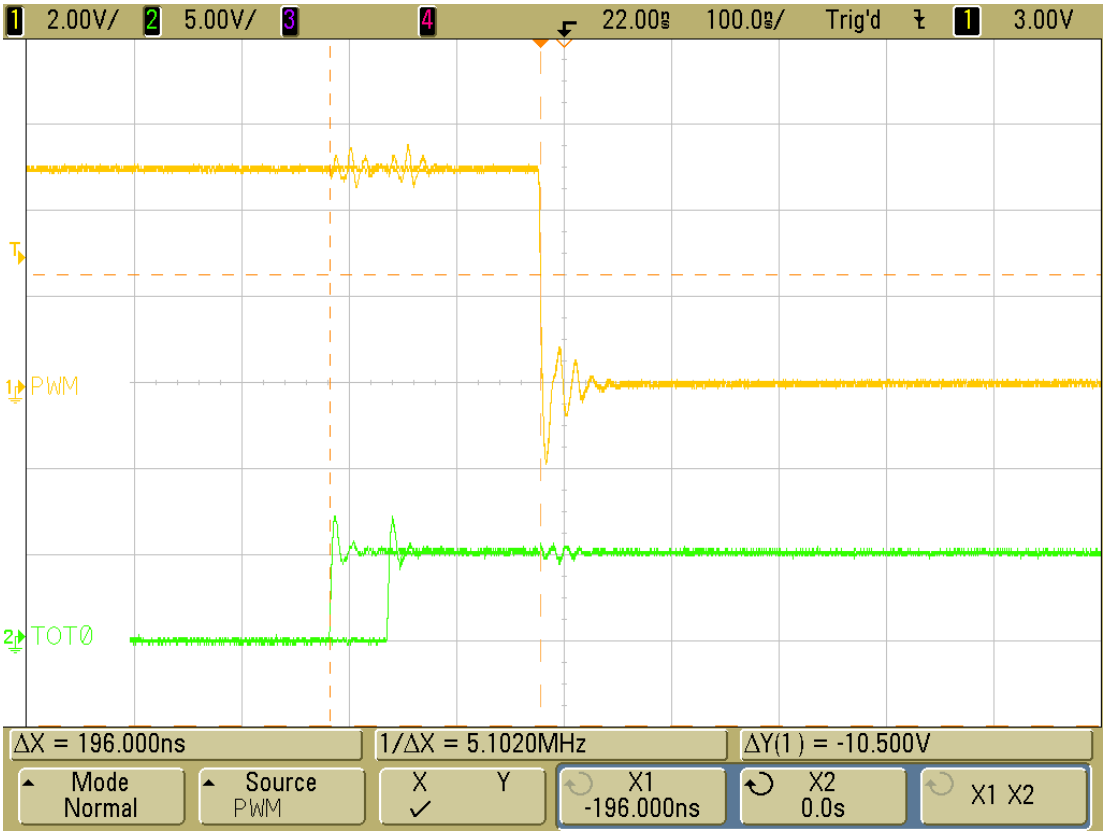


Figure 11. Falling Edge for example 2 – Delay



4.1.2 Influence of CPU access on peripheral bus

To measure the influence of CPU accesses to the peripheral bus, a small code writing to and reading from another port data register is implemented inside the while-loop of previous example. This short example represents bus accesses as they may be implemented in application. There are always short releases of the bus as instructions to update registers or branch instruction are implemented. 100% access to the peripheral bus (only MOV A,I:09 instruction) is absolutely unlikely.

See C source code and generated assembly code below:

```
234:   while(1)
   235:   {
   236:     buf = PDR09;
FC02F5: 5009              MOV     A,I:09      <-- Write to peripheral bus
FC02F7: 98                MOVW    RW0,A
   237:     buf++;
FC02F8: 3001              ADD     A,#01
FC02FA: 98                MOVW    RW0,A
   238:     PDR09 = buf;
FC02FB: 5109              MOV     I:09,A      <-- Read from peripheral bus
   239:   }
FC02FD: 60F6              BRA     FC02F5
   240: }
```

Figure 12 and Figure 13 show the rising edge of the PWM signal in correlation to the reload timer output signal. You can see again same jitter of +36ns like without the CPU access to the peripheral bus, but now the delay itself is increased to 194ns (230ns-36ns, ~ 13 CPU cycles) due to blocked internal busses by CPU access.

Regarding the long period time of PWM signal and reload timer cycle time, again this value is negligibly small.

Figure 12. Rising Edge with CPU influence – Jitter

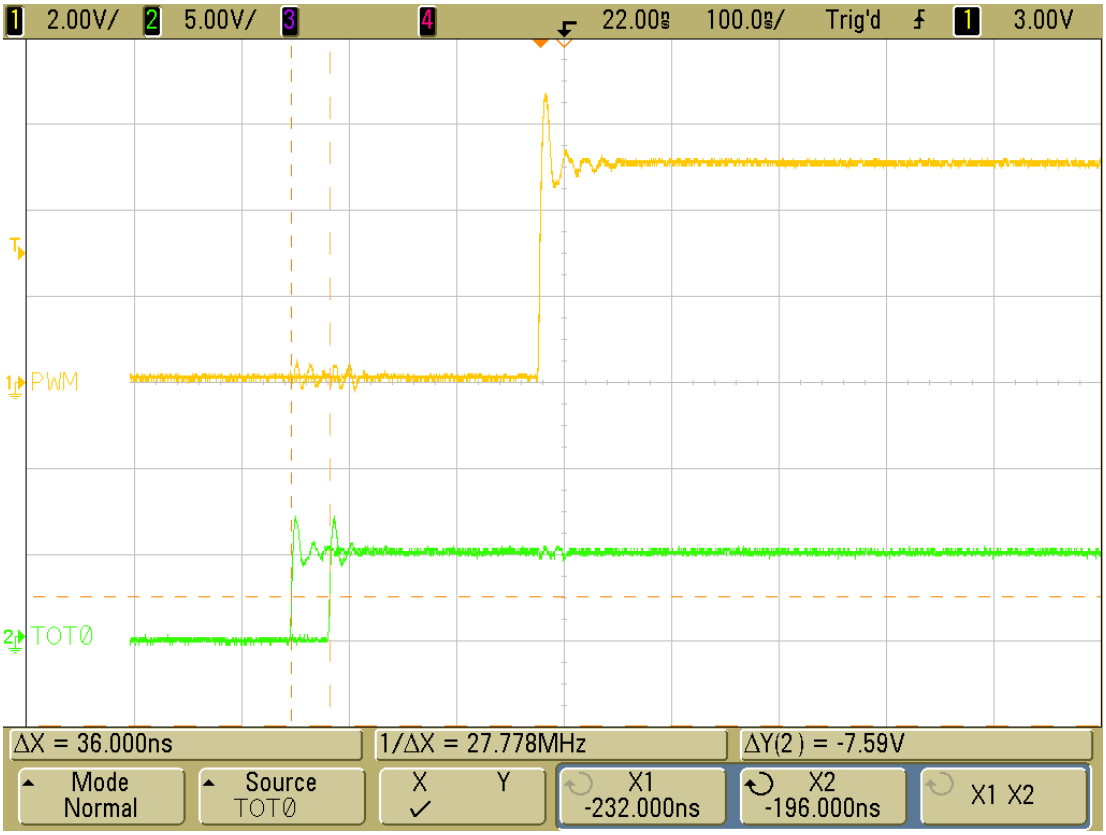
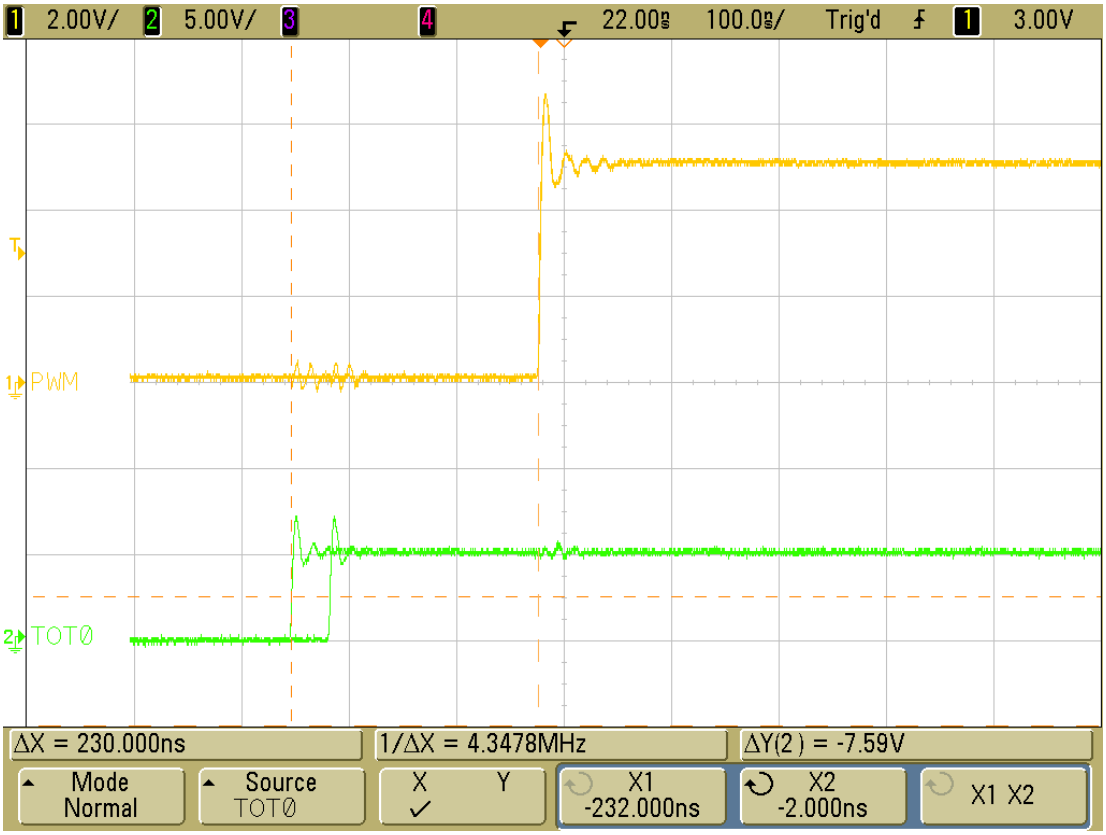


Figure 13. Rising Edge with CPU influence – Delay



4.1.3 Influence of other DMA channel operations

The 16FX Family offers up to 16 DMA transfer channels that can be used independently. The channels have different priorities for requesting the internal busses, starting from channel 0 with highest priority down to channel 16 with lowest priority.

If two resources ask for a DMA transfer at the same time, DMA channel with lower number and therefore higher priority will get access to the bus. Nevertheless, the higher prior DMA channel has to wait a currently running lower prior DMA transfer to be ready. As on 16FX maximum amount of data to be transferred at one time is only a 16bit word, delay is very short (ideally two CPU cycles).

Therefore you have to analyze your system regarding DMA transfer with different priorities if you phase problems with the accuracy of the PWM signal. For best accuracy PWM DMA channel has to be highest priority. In this case only CPU request for the bus is more important.

4.1.4 Influence of other interrupts

Different interrupt levels may also have an impact on the accuracy of the PWM signal. Within a PWM cycle the interrupt level does not matter, as the interrupt requests are only passed to the DMA controller to trigger a transfer, but at the end of one PWM period the interrupt request is forwarded to the CPU, which exploits the level of an incoming interrupt request before executing the adequate interrupt service routine.

So interrupts with higher priority or currently running interrupt service routines of interrupts with the same priority may delay the execution of the interrupt service routine to re-initialize the DMA transfer for the PWM. These interrupt service routines are no problem if the execution of the PWM ISR (or rather the reload timer interrupt service routine) is guaranteed within the time needed for one run of the reload timer (reload timer cycle time).

Interrupts with lower priority are not problematic as they are interruptible by the PWM ISR.

4.2 Influence on CPU operation

There should be hardly any influence of the software PWM generation on CPU execution as the DMA transfer from the PWM table in RAM or ROM to IO register is executed in parallel to CPU instruction execution. Only after a full PWM cycle when the predefined number of DMA transfers is finished, CPU operation is shortly interrupted for execution of the interrupt service routine. Here the DMA is re-initialized and interrupt flags are cleared.

To measure the influence of the software PWM generation on CPU performance, a simple implementation of Dhrystone benchmark V2.1 is used. Please keep in mind that the final benchmark results are not optimized ones! Benchmark is executed with not optimized standard settings just to show impact of the software PWM on performance.

First Dhrystone test is run without generating software PWM. See logfile of benchmark output:

```
DHRYSTONE BENCHMARK V2.1 FOR MB96F346RSA - SW-PWM OFF !!!

Dhrystone Benchmark, Version 2.1 (Language: C or C++)
Register option not selected

        1000 runs   0.04 seconds
       10000 runs   0.39 seconds
       20000 runs   0.79 seconds
       40000 runs   1.57 seconds
       80000 runs   3.14 seconds
      160000 runs   6.28 seconds

Final values (* implementation-dependent):

Int_Glob:      O.K.  5  Bool_Glob:     O.K.  1
Ch_1_Glob:     O.K.  A  Ch_2_Glob:     O.K.  B
Arr_1_Glob[8]: O.K.  7  Arr_2_Glob8/7: WRONG      28938
Ptr_Glob->              Ptr_Comp:       *    17234
  Discr:       O.K.  0  Enum_Comp:     O.K.  2
  Int_Comp:    O.K.  17 Str_Comp:      O.K.  DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->         Ptr_Comp:       *    17234 same as above
  Discr:       O.K.  0  Enum_Comp:     O.K.  1
  Int_Comp:    O.K.  18 Str_Comp:      O.K.  DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:     O.K.  5  Int_2_Loc:     O.K.  13
Int_3_Loc:     O.K.  7  Enum_Loc:      O.K.  1
Str_1_Loc:                             O.K.  DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:                             O.K.  DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:      39.25
Dhrystones per Second:                           25476
VAX  MIPS rating =                               14.50

DHRYSTONE END !!!
```

Secondly, Dhrystone test was run with software PWM generated in background. Settings for PWM are the same as used in example 2 in chapter 4.1.1.

```
DHRYSTONE BENCHMARK V2.1 FOR MB96F346RSA - SW-PWM ON !!!

Dhrystone Benchmark, Version 2.1 (Language: C or C++)
Register option not selected

        1000 runs   0.04 seconds
       10000 runs   0.40 seconds
       20000 runs   0.79 seconds
       40000 runs   1.58 seconds
       80000 runs   3.16 seconds
      160000 runs   6.32 seconds

Final values (* implementation-dependent):

Int_Glob:      O.K.  5  Bool_Glob:     O.K.  1
Ch_1_Glob:     O.K.  A  Ch_2_Glob:     O.K.  B
Arr_1_Glob[8]: O.K.  7  Arr_2_Glob8/7: WRONG      28938
Ptr_Glob->             Ptr_Comp:        *    17234
  Discr:       O.K.  0  Enum_Comp:     O.K.  2
  Int_Comp:    O.K.  17 Str_Comp:      O.K.  DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->        Ptr_Comp:        *    17234 same as above
  Discr:       O.K.  0  Enum_Comp:     O.K.  1
  Int_Comp:    O.K.  18 Str_Comp:      O.K.  DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:     O.K.  5  Int_2_Loc:     O.K.  13
Int_3_Loc:     O.K.  7  Enum_Loc:      O.K.  1
Str_1_Loc:                             O.K.  DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:                             O.K.  DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:        39.53
Dhrystones per Second:                          25297
VAX  MIPS rating =                                14.40

DHRYSTONE END !!!
```

Comparing these results, you can see that in second case there is loss of performance of around 0.7%. This value is of course depending on the PWM frequency, but you can see that impact on CPU performance is very low.

5 Appendix

5.1 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software example related to this application note is:

96340_sw_pwm_rlt_dma_io

It can be found on the following Internet page:

http://www.cypress.com/cypress-mcu-product-softwareexamples

6 Document History

Document Title: AN204827 - F²MC-16FX Family, MB96340 Software PWM by use of DMA transfer

Document Number:002-04827

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	09/20/2007	Initial release
			04/02/2009	Some changes because of updated software example
*A	5086024	NOFL	05/31/2016	Migrated Spansion Application Note MCU-AN-300239-E-V11 to Cypress format

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Lighting & Power Control	cypress.com/powerpsoc
Memory	cypress.com/memory
PSoC	cypress.com/psoc
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless/RF	cypress.com/wireless

PSoC® Solutions

cypress.com/psoc

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums |Blogs | Video |Training

Technical Support

cypress.com/support

PSoC is a registered trademark and PSoC Creator is a trademark of Cypress Semiconductor Corporation. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2007-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.