APPLICATION NOTE

USING ST7 PWM SIGNAL TO GENERATE ANALOG OUTPUT (SINUSOID)

by Microcontroller Division Application Team

INTRODUCTION

The purpose of this note is to present how to use the ST7 PWM/BRM for the generation of a 50Hz sinusoïd tunable in average and amplitude. Our application has been done with a ST72511R4.

1 ST7 PWM/BRM GENERATOR

In this part the main PWM/BRM features of the ST7 are pointed out. Please refer to the ST7 datasheet for more details.

The ST7 PWM/BRM includes a 6-bit Pulse Width Modulator (PWM) and a 4-bit Binary Rate Multiplier (BRM) Generator. It allows the digital to analog conversion (DAC) when used with external filtering.

PWM GENERATION

The counter increments continuously, clocked at internal CPU clock. Whenever the 6 least significant bits of the PWM counter overflow, the output level for all active channel (only PWM0 in this application) is set.

When a match occurs between the PWM counter and the PWM binary weight, the corresponding output level is reset. (see Figure 1).

This PWM signal must be filtered with an external RC network selected for the filtering level required.

Dedicated pins for the PWM/BRM are connected to a 1k serial resistor which must be taken into account to calculate the RC filter time.(see Figure 2).

In any case, the RC filter time must be higher than T_{cpu} x64 (= 8µs here because f_{cpu}=8MHz).

In this application, no additional external resistor is used; the value of C_{ext} used is 1µF.

The RC filter time is then equal to 1ms (it has to be higher T_{cpu}x64).

Figure 1. PWM Generation

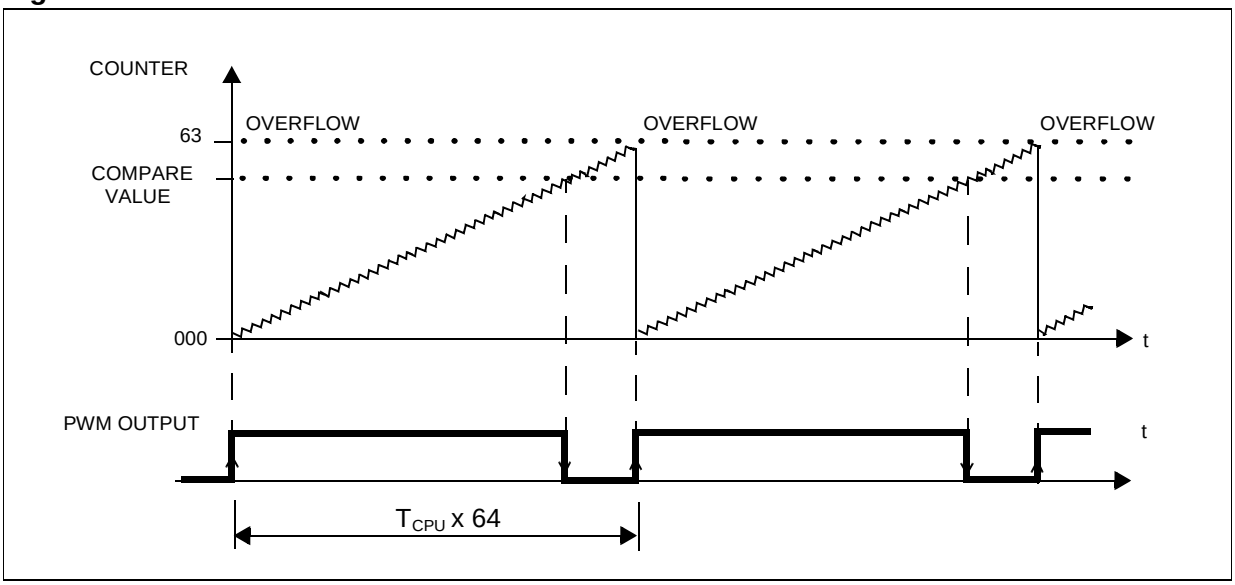
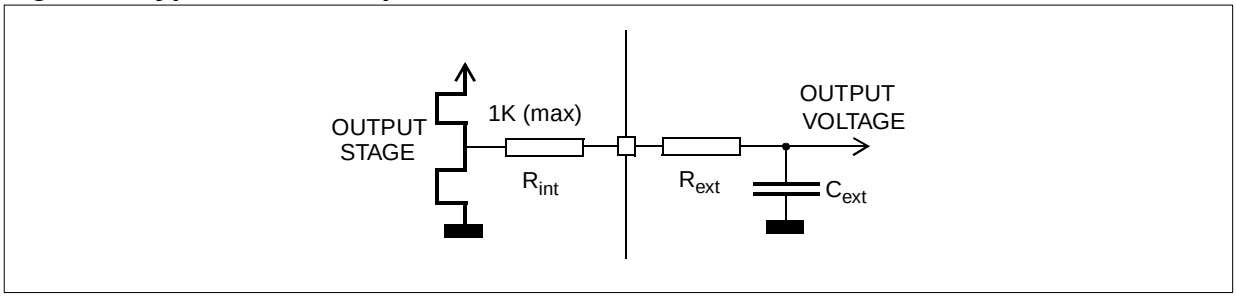


Figure 2. Typical PWM Output Filter



BRM GENERATION

The BRM bits allow the addition of a pulse to widen a standard PWM pulse (with duration of T_{cpu}) for specific PWM cycles. This has the effect of “fine-tuning” the PWM Duty cycle (without modifying the base duty cycle), thus, with the external filtering, providing additional fine voltage steps (see Figure 3).

The PWM intervals which are added to are specified in the 4-bit BRM register and are encoded as shown in the following table. The BRM values shown may be combined together to provide a summation of the incremental pulse intervals specified (see Figure 4).

Table 1. Bit BRM Added Pulse Intervals (Interval #0 not selected).

BRM 4 - Bit Data	Incremental Pulse Intervals
0000	none
0001	i = 8
0010	i = 4,12
0100	i = 2,6,10,14
1000	i = 1,3,5,7,9,11,13,15

Figure 3. BRM pulse addition (PWM > 0)

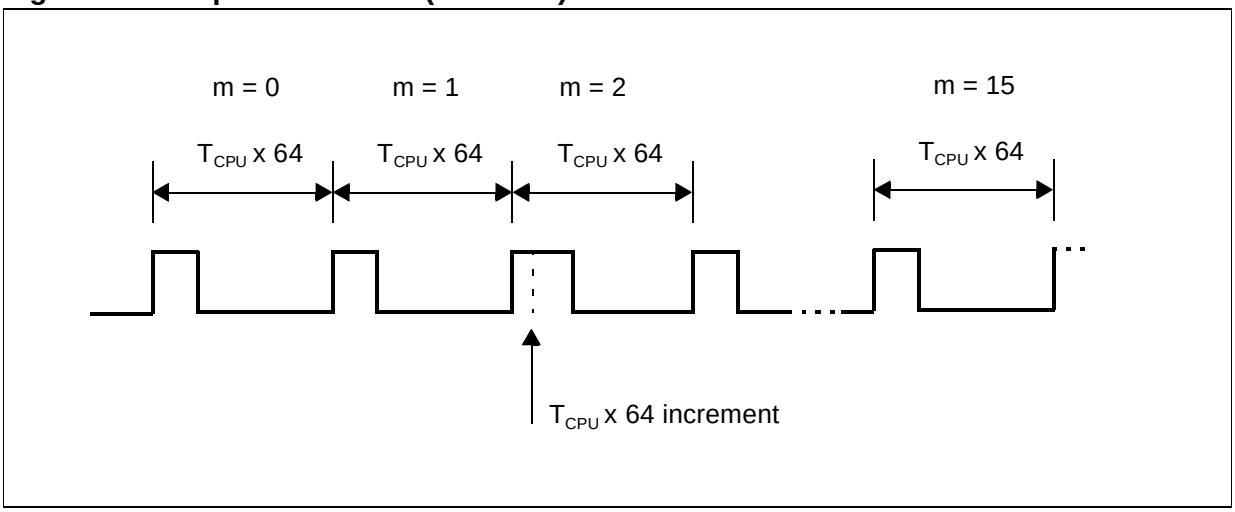
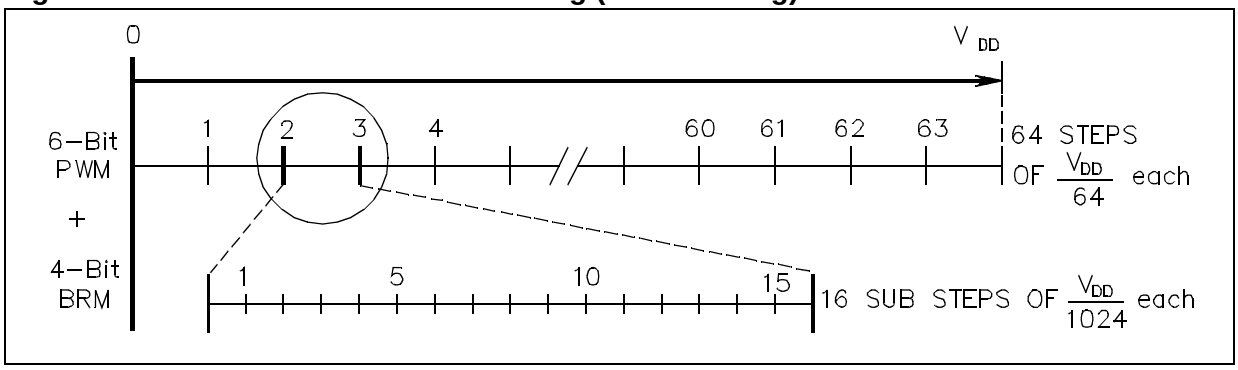


Figure 4. Precision for PWM/BRM Tuning (after filtering).



REGISTER DESCRIPTION

The 10 bits are separated into two data registers:

PULSE BINARY WEIGHT REGISTER

7							0
1	POL	P5	P4	P3	P2	P1	P0

Channel 2 Pulse Binary Weight Register (PWM2)

Channel 3 Pulse Binary Weight Register (PWM3)

Bit 7 = Reserved (Forced by hardware to “1”)

Bit 6 = **POL** *Polarity Bit for channel i.*

0: The channel *i* outputs is a “1” level during the binary pulse and a “0” level after.

1: The channel *i* outputs is a “0” level during the binary pulse and a “1” level after.

Bit 5:0 = **P[5:0]** *PWM Pulse Binary Weight for channel i.*

This register contains the binary value of the pulse.

BRM REGISTERS

7							0
B7	B6	B5	B4	B3	B2	B1	B0

Channels 3+2 BRM Register (BRM32)

Bit 7:4 = **B[7:4]** *BRM Bits (channel i+1).*

Bit 3:0 = **B[3:0]** *BRM Bits (channel i)*

This register defining the intervals where an incremental pulse is added to the beginning of the original PWM pulse. Two BRM channel values share the same register.

From the programmer's point of view, the PWM and BRM registers can be regarded as being combined to give one data value.

For example：

1	POL	P	P	P	P	P	P	+	B	B	B	B

Effective (with external RC filtering) DAC value

1	POL	P	P	P	P	P	P	B	B	B	B

2 A 50HZ SINUSOID

The goal of this application is to generate a 50Hz sinusoïd. The use of the BRM allows us to have a better precision thanks to the sub steps it creates (Vdd/1024).

In this application, there are two ways to obtain these values:

- you can call the function calc() at the beginning of the main program (this function calculates the 64 desired values one time and stores them into a table called value[] in RAM).

- you can also use the table value[] declared in ROM with all the 64 values, calculated before running the application. If you change the amplitude or the offset of the signal, take care to change values into the table (file table.c).

According to the way you choose, you have to include the files function.c and function.h or table.c and table.h (please, refer to the listing of the program: the unchosen way is in comment).

These values are words (16 bit-long). The four least significant bits are put in BRM and the following six bits are put in PWM. Effectively,the biggest value is 3FF (got for cos(0) recentered in [0..1023]), the six upper bits are then unused because at zero.

We use an hardware Watchdog, active just after reset; for this reason, it has to be refreshed every 98ms (or less).

To describe a sinusoïd between 0 and 2 π, we used 64 values with steps of $2\pi/64$ radians.

FREQUENCY

To have the desired frequency (50Hz here), a real time base is created using a 16 bit timer output compare interrupt (see AN974). The period is 20ms, and then 625 counts timer (decimal value) are needed (fcpu/4=2MHz, and there are 64 points to describe the sinusoïd). During interrupts, a value is put into the BRM and the PWM registers and the OC1R counter is updated.

Here follows a table giving different values of the counter for different frequencies:

f(Hz)	50	60	70	80	90	100
Counter value	625	521	446	391	347	313

OCCUPATION TIME

During an interrupt and for this program running at fcpu=8MHz, the CPU is used at 11.76% (36.75µs) by loop of 20ms/64 (because the frequency considered is 50Hz). That means that 88.24% of the CPU are free to do anything else.

OFFSET AND AMPLITUDE

The function calc() calculates the sinusoïd values with the expression:

$$Am*\cos(X)+offset$$

where Am represents the amplitude of the sinusoïd and offset the way to change the average value. As the sinusoïd has to be between 0 and 1023 (0 and 5 Volts) for an efficient use of the 10 bit DAC (2^{10}=1024), initial values for Am and offset in this application are 511 and 512 (to have the maximum amplitude).

These values can be changed but there are the following constraints:

$$offset+Am \leq 1023$$
$$offset-Am \geq 0$$

3 FLOWCHARTS

Here follows the flowchart of the main program using the function calc():

Figure 5. Main Program Flowchart

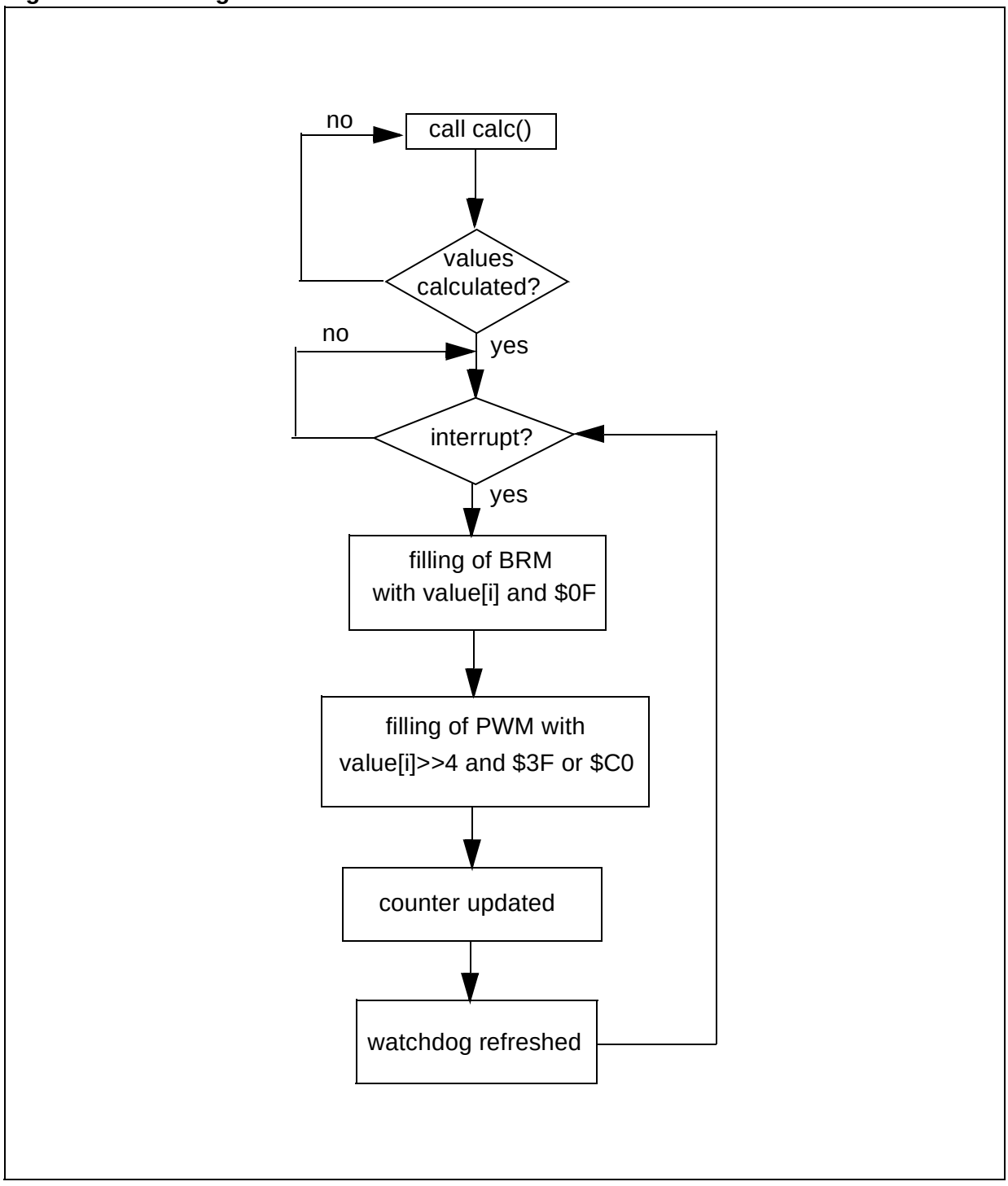
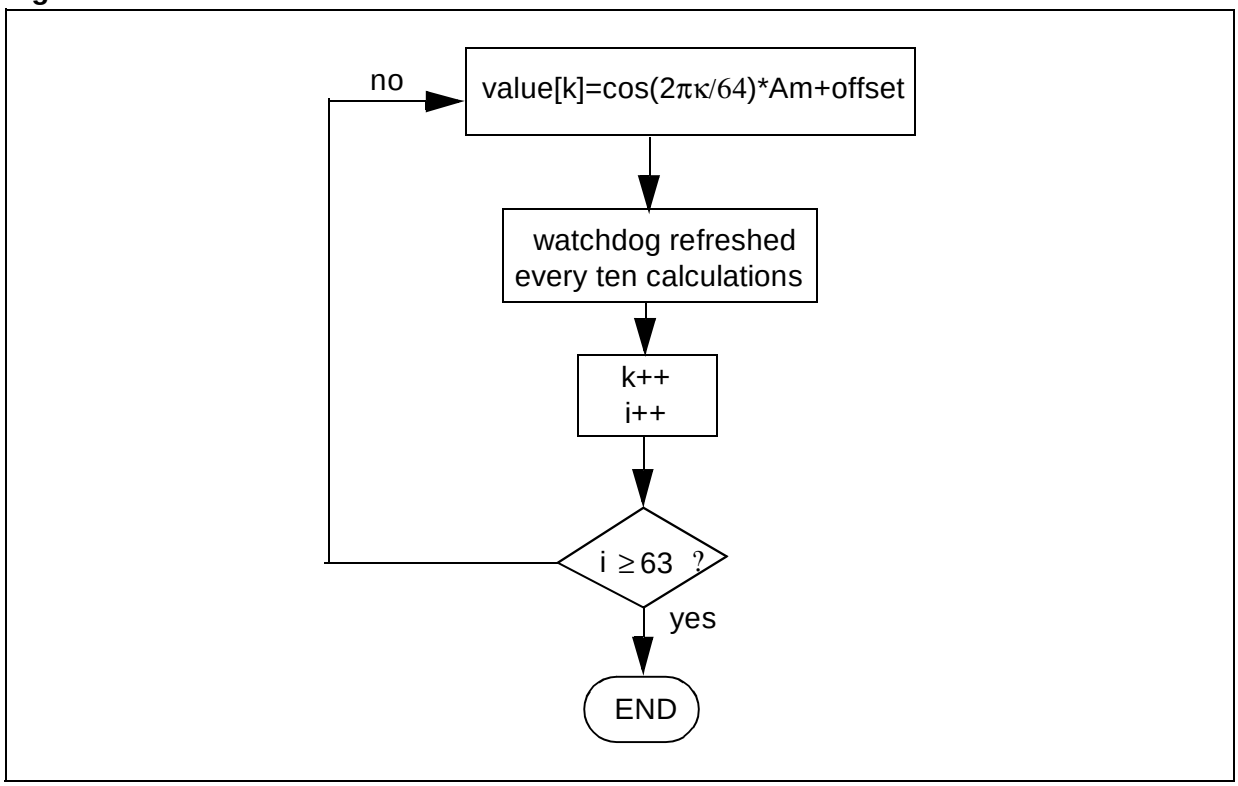


Figure 6. Function Flowchart



4 SOFTWARE

The assembly code given below is for guidance only. For missing label declaration please refer to the register label description of the datasheet or the ST web software library ("map7250 .c" file...).

main.c:

```
// Include files

#include "map7250.h"   // ST7250 memory and registers mapping
#include "variable.h"  // Define your global variables here
#include "function.h"  // All functions used in the application can be defined
                       / here for a good project management.
//#include "table.h"

//-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

void main(void)
{
 PWM2=0x80;                 // Initialization of PWM and BRM registers.
 BRM32=0x00;

 calc();
                     // Call of the function calculating the sinusoïd values.
                     // This takes a long time, it's better to use the table values
                     // located in ROM (see file table.c).
asm {
    sim              // Disable all interrupts.
   }
 TAOC1HR = (delta>>8);       // Load the compare value in OC1R.
 TAOC1LR = delta;
 TACLR = 0;                  // Reset the timer at FFFC.
 TACR1 = 0x41;               // Timer A in Output Compare with no other interrupt, OLV1 set.
 TACR2 = 0x80;               // Timer clock=fcpu/4->0.5µs in normal mode with a 16MHz quartz.

 asm {
    rim              // Enable interrupts.
   }


 while(1)
    {WDGCR=0x7F;}     // Refresh of the Watchdog (because of the reset every 98ms).

}

/*** (c) 1998 STMicroelectronics ****************** END OF FILE ***/
```

itpwmb.c: (interrupt routine)

```
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

DESCRIPTION :  Main Interrupt Service Routines

        This file can be used to describe all the interrupt subroutines
        that may occur within your application.
        As the routines' names are declared in the .prm file, when an interrupt
        happens, the software will branch automaticly to the corresponding routine
        according to the interrupt vector loaded in the PC register.
        For now, the routines are all empty as nothing special may occur in
        the example's main program.
        It's better to create as many interrupt functions as necessary, ie for each
        interrupt vector even if the routine is empty (iret). By doing this, you
        will prevent your software from branching to an undesired interrupt routine.

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

MODIFICATIONS :

******************************************************************************/

#include "map7250.h"
#include "variable.h"
#include "lib_bits.h"

#pragma TRAP_PROC SAVE_REGS
/*-----------------------------------------------------------------------------
ROUTINE NAME : tima_rt
INPUT/OUTPUT : None

DESCRIPTION  : timer Interrupt Service Routine

COMMENTS   :
-----------------------------------------------------------------------------*/

void tima_rt(void)
{
 #pragma DATA_SEG _ZEROPAGE
 static unsigned char index=0;
 unsigned int  total;

 #pragma DATA_SEG DEFAULT
 if( ValBit(TASR,6) )              // First step to clear OCF1. Test if the IT is generated by
OCF1.
{
 BRM32 = value[index] & 0xF;          // The four least significant bits are put in the BRM
register.
 PWM2=((value[index] >> 4) & 0x3F) | 0x80;  // Bit 7=1 (unused), POL=0.
```

```
                        // The six upper bits of the table value are put in PWM0(counter).
 if(index==63) index=0;            // If it's the end of the table, jump to its begin.

 total=(TAOC1HR<<8)+TAOC1LR;    // Total of the counter (16 bits). Second step to clear the
OCF1 flag.
 total+=delta;           // Update of the counter.
 TAOC1HR=(total>>8);        // Update of TAOC1HR.
 TAOC1LR=total;          // Update of TAOC1LR.

 index++;             // Next value.
}
else
{
  TAOC2LR=0x0;          // Second step to clear the OCF2 flag.
}
/*** (c) 1998 STMicroelectronics ***************** END OF FILE ***/
```

SOFTWARE

"THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNEXION WITH THEIR PRODUCTS."