

## F2005 Abstract

The described system is applied to get the German homologation certificate for an adjustable rear muffler of a motorbike.

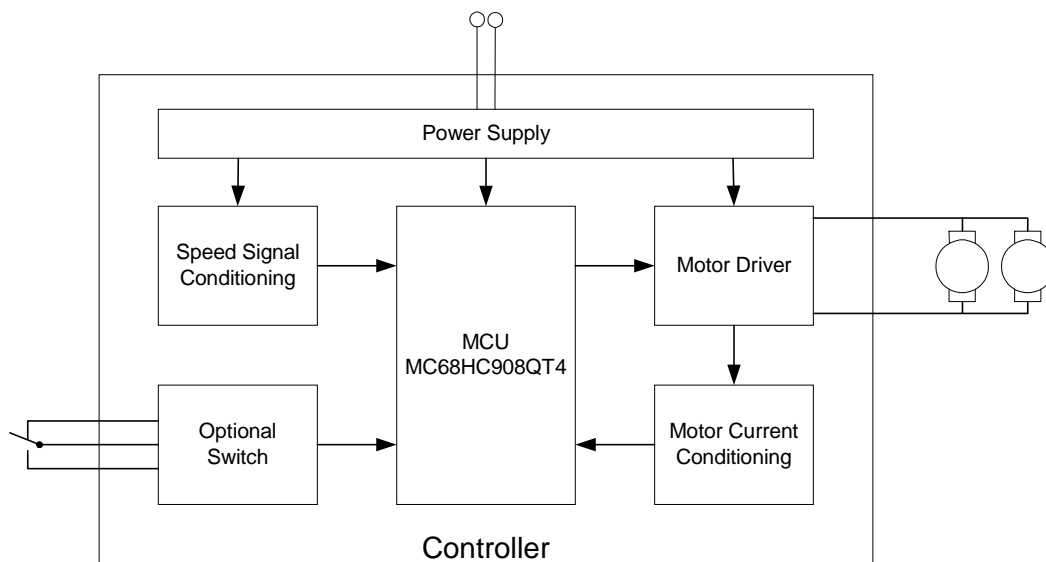
It is possible to bypass the restrictive law about noise pollution by a trick.

The procedure of measuring the sound intensity is precisely defined. The sound intensity is measured in a defined vehicle speed range. The bike passes a starting point at a defined speed. Afterwards, it is accelerated on a defined distance. This procedure is performed in two gears. The trick is to reduce the emitted noise around the prescribed test speed range when the bike is accelerated. Below and above that defined speed range noise isn't of interest for the law.

To vary the emitted sound level of a motorbike, there are adjustable rear mufflers.

They provide legal loud exhaust system noise on the one hand and real good (but unfortunately illegal loud) sound on the other hand. There are manually and electromotively operated adjustors as well.

The task of the described system is to adjust an electromotively actuated muffler in a way that during the homologation run the emitted noise stays under the legal threshold.







## Software Listing

```
/*  
** #####  
**  
** Filename : Events.C  
** Project  : Contest_F2005  
** Processor: MC68HC908QT4CP  
**  
** Beantype : Events  
**  
** Version  : Driver 01.00  
**  
** Compiler : Metrowerks HC08 C Compiler V-5.0.16  
**  
** Date/Time : 05.03.2003, 15:56  
**  
** Abstract :  
**  
**   This is user's event module.  
**   Put your event handler code here.  
**  
** Settings :  
**  
** Contents :
```

```

**
**      Cpu_OnSwINT - void Cpu_OnSwINT(void);
**
**
**      (c) Copyright UNIS, spol. s r.o. 1997-2002
**
**      UNIS, spol. s r.o.
**      Jundrovska 33
**      624 00 Brno
**      Czech Republic
**
**      http   : www.processorexpert.com
**      mail   : info@processorexpert.com
**
** #####
*/
/* MODULE Events */

#pragma MESSAGE DISABLE C1420 /* WARNING C1420: Result of function-call is ignored */

/*Including used modules for compiling procedure*/
#include "Cpu.h"
#include "Events.h"
#include "RPM.h"
#include "Current.h"
#include "Switch.h"
#include "Driver.h"

/*Include shared modules, which are used for whole project*/
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

/*****
* Constants
*****/
#define CAPTURE_TIME_MIN          0          // [CapTMax>>16]
#define CAPTURE_TIME_MAX          65535     // [CapTMax>>16]
#define CAP_DELTA                  5         // [100/x %]
#define GET_FIRST_VALUE           0         // State machine status
#define GET_SECOND_VALUE          1         // State machine status
#define GET_LAST_VALUE            2         // State machine status

/*****
* Variables
*****/
extern word current;                // Driver load current [mA]
extern word rev_period;             // Represents current engine rpm
// [CapTMax>>16]
extern byte status;                // Universal status variable
extern byte cap_control;           // Counter to check capture events

/*
** =====
**      Event   : Current_OnEnd (module Events)
**
**      From bean : Current [ADfast]
**      Description :
**      This event is called after a measurements (which consists
**      of 1 or more conversions) is finished.
**      Parameters : None
**      Returns   : Nothing
** =====
*/
void Current_OnEnd(void)
{
    // Get driver load current

```

```

do
{
    status = Current_GetValue16(&current);
} while (status != ERR_OK);

// Formula for conversion from ADC value to current:
// current [mA] = (current >> 8) * (1/Rshunt[Ohm] * 5000mV/255)
current = (current >> 8) * 10; // with Rshunt = 2 Ohm
}

/*
** =====
** Event    : RPM_OnCapture (module Events)
**
** From bean : rev_period [Capture]
** Description :
**   This event is called on capturing of Timer/Counter actual
**   value.
** Parameters : None
** Returns    : Nothing
** =====
*/
void RPM_OnCapture(void)
{
    /*****
    * Description of working method
    *
    * The engine speed can be determined by a HF-burst signal which
    * appears about every 50ms on the 12V-DC.
    * Because of the engine's inertia and the relatively high frequency,
    * the HF-frequency is expected to be constant during one burst.
    * To determine the current engine speed with adequate accuracy,
    * three oscillations from every HF-burst are evaluated. The result
    * is the average of three plausible subsequent measured values.
    * The first measured value is taken as reference value for the
    * second one. If the second value fits in a defined window around
    * the first value, it is regarded as plausible value.
    * Afterwards the average of the two values is calculated.
    * The second measured value must fit in a window around the
    * average value. The average of all three values is regarded as the
    * correct period of the current engine speed proportional HF-signal.
    * If the second or third measured value doesn't fit to its
    * predecessor, it is regarded as new first value and procedure
    * starts from the beginning.
    * The method is very reliable, because it is very unlikely to
    * measure three subsequent wrong periods of almost the same value.
    *****/
    word cap_val;           // Current capture value [CapTMax>>16]
    static word cap_average; // Average capture value [CapTMax>>16]
    static byte cap_status; // State machine status

    // Get engine revs related ignition-pulse-interference-signal-period
    // (WOW - what an expression !!)
    do
    {
        status = RPM_GetCaptureValue(&cap_val);
    } while (status != ERR_OK);

    // Three consecutive matching values are averaged
    switch (cap_status)
    {
        case GET_FIRST_VALUE:
            // Check if capture value is of interest
            if ((cap_val > CAPTURE_TIME_MIN) &&
                (cap_val < CAPTURE_TIME_MAX))
            {
                cap_average = cap_val;
                cap_status = GET_SECOND_VALUE;
            }
    }
}

```

```

        break;

    case GET_SECOND_VALUE:
        // Check if second capture value fits to first value
        if ((cap_val > (cap_average - (cap_average / CAP_DELTA))) &&
            (cap_val < (cap_average + (cap_average / CAP_DELTA))))
        {
            cap_average = (cap_average >> 1) + (cap_val >> 1);
            cap_status = GET_LAST_VALUE;
        }
        else
        {
            // Second capture value out of tolerance band -> Reset
            cap_status = GET_FIRST_VALUE;
        }
        break;

    case GET_LAST_VALUE:
        // Check if third capture value fits to first two values
        if ((cap_val > (cap_average - (cap_average / CAP_DELTA))) &&
            (cap_val < (cap_average + (cap_average / CAP_DELTA))))
        {
            cap_average = (cap_average >> 1) + (cap_val >> 1);
            rev_period = cap_average; // new rpm period determined
        }
        cap_status = GET_FIRST_VALUE;
        break;

    default:
        break;
}

// Reset counters
RPM_Reset();
cap_control = 0;
}

/* END Events */

/*

```