HCS08 Unleashed: Designer’s Guide To The HCS08 Microcontrollers

Errata

27/08/2010

This errata file contains the most recent updates and errors found in the book. Most of the following itens are not errors but clarifications and important additional information that are not part of the first edition of the book. Modified text appears highlighted in yellow.

Thanks for all the valuable feedbacks (Daniel Friederich, Jim Donelson, Ewen McNaught (aka BigMac), Tsuneo Chinzei, Marcelo Ribeiro, etc.).

Page 38: The last example should be changed to:

```
          -40          0xD8
  +      -100          0x9C
        ------      ----------
         -140          0x74   <---   Overflow !
V = 1                                0x74 = +116 !
```

Page 65: C examples using the LAP register should be changed to:

Examples:

2. Accessing a variable stored at 0x10000 (in C):

```
LAP0 = 0; LAP1 = 0; LAP2 = 1;  // LAP = 0x010000
temp = LB;                     // load temp with the byte value pointed by LAP
```

Or, using the special macros defined in the “mmu_lda.h” file:

```
__LOAD_LAP_ADDRESS(variable);  // the address of variable is stored into LAP
temp = LB;                     // load temp with the byte value pointed by LAP
```

4. Accessing a byte array on paged memory (include the “mmu_lda.h” file):

```
// For arrays up to 127 bytes
__LOAD_LAP_ADDRESS(array_name); // the address of the array is stored into LAP
LAPAB = index;       // for arrays with up to 127 bytes
temp = LB;           // temp = array[index] value

// For arrays larger than 127 bytes:
__LOAD_LAP_ADDRESS(array_name); // the address of the array is stored into LAP
__ADJUST_LAP_IMM_16BIT(index);
temp = LB;           // temp = array[index] value
```

Page 68: Table 2.10: the memory sizes for the GT8A and GT16A should be read as follows:

Model	V_{DD}	FLASH (KiB)	RAM (byte)	EEPROM (byte)	BUS CLK (MHz)	I/O	Timers / CCP channels	IR	SCI	USB	CAN	I^2C	SPI	ADC (channels/ bits)	AC	Package
GT8A	L	8	1,024	-	20	up to 39	2 – 16 bits / 3+2 CCP	-	1	-	-	1	1	8 / 10 bits	-	QFN32, SDIP42, QFP44, QFN48
GT16A	L	16	2,048	-	20	up to 39	2 – 16 bits / 3+2 CCP	-	1	-	-	1	1	8 / 10 bits	-	QFN32, SDIP42, QFP44, QFN48

Page 79: The text regarding memory model should be read as follows:

The memory model is related to how the compiler and the linker arrange code and data in memory:

1. On the tiny memory model, all data (including the stack) is stored in the direct page, unless it is declared by using **#pragmas** and the **__far** qualifier. This model allows faster access to variables by using the direct addressing mode.

2. On the small memory model, data can reside anywhere within the 64 KiB address space by using 16-bit addresses (instead of the 8-bit addresses of the tiny memory model). This model is slower than the tiny one because it uses the extended addressing mode. On the other hand, it allows access to the full range of RAM memory available on the device. Using the direct memory area is only possible by the use of **#pragmas** (such as **#pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE**) and the **__near** qualifier.

Page 81: The first entry of table 3.1 should be read as follows:

File Extension	Description
*.ABS	Absolute file generated by the linker tool

Page 85: Text following listing 3.2 should be read as follows:

Notice that the areas defined on the "SEGMENTS" section can be modified by the user to better suit a specific use. Typical situations are:

1. Modifying Z_RAM and RAM area to increase available area for extended addressing mode on small memory model:

```
// Reserves 0x20 bytes for direct addressing mode. The remaining RAM (0x0080
// to 0x025F) is accessed using extended mode:
Z_RAM =  READ_WRITE   0x0060 TO 0x007F;
RAM   =  READ_WRITE   0x0080 TO 0x025F;
```

2. Modifying ROM area to leave space (one or more pages) for storing non-volatile data. This is done by reducing the size of the ROM segment:

```
// Reserves the first 256 bytes of flash for non-volatile data (addresses
// from 0xE000 to 0xE0FF will not be used for code-storage):
ROM   =  READ_ONLY    0xE100 TO 0xFFAD;
```

It is also possible to include the Z_RAM into the DEFAULT_RAM. This enables the linker to place variables into the RAM segment and once it is full, the Z_RAM segment is filled as needed:

```
DEFAULT_RAM       =  INTO RAM, Z_RAM;
```

Page 111: There is an error on the comment on the code example, item 4.1.3.2:

```
char v4 = 010;  // declares v4 and initializes with 8 decimal
```

Page 115: The third sample program in that page should be read as follows:

Notice that on expressions, numeric constants always default to the lowest possible type. This can lead to an unexpected behavior, for example:

```
unsigned long int a;
unsigned int b;
a = b * 1000;
```

Page 117: The second paragraph should be read as follows:

This line declares two **char** pointer variables. By default, on the HCS08, each pointer occupies two bytes of memory (because addresses are 16-bit), that is why the variable p2 on figure 4.1 is located at address 0x0102 (two bytes after p1). Note that **__near** pointers use 1-byte address thus occupying only one byte; on the other hand, **__linear** pointers occupy three bytes (because they point to 24-bit addresses).

Page 120: The third line and the box of topic 4.1.5.5 should be read as follows:

For HCS08 devices, the compiler must follow the big endian model for storing the elements of the union.

ⓘ *On the little endian format, variables larger than a byte are stored from the lowest byte to the highest byte (the highest address stores the highest significant byte); whereas on the big endian format, the storage is done from the highest to the lowest byte (the highest address stores the lowest significant byte). Little endian = LSB first, Big endian = MSB first!*

Page 126: Table 4.8 should be read as follows:

@	**__far**	**__near**	**__linear**	**LINEAR**
__alignof__	**__va_sizeof__**	**__interrupt**	**__asm**	

Table 4.8 – C language extensions

Page 127: The correct keywords are __far, __near, __interrupt and __asm. The text should be changed as follows:

__far - this keyword tells the compiler to use 16-bit addresses when accessing the variable, allowing access to variables located in any of the 65536 possible addresses. When used on pointer declarations this keyword acts as an access modifier, affecting the last variable declared on its left. The hidef.h file also creates the **far** alias for **__far**. Examples:

```
// declare a far pointer to a far pointer to a char
char *__far *__far my_far_pointer;
// declare a pointer to a far pointer to an int
int *__far *my_far_pointer;
```

It is also possible to use this keyword with arrays. This can be done as follows:

```
// declare the function parameter as an element of the "param" array
void test_function (char param[2] __far);
// or by using a single pointer:
void test_function (char *__far param);
```

__near - the near keyword is used to allow faster addressing on variables stored into the direct page. It works in the opposite way of the far keyword. The compiler also supports the **near** alias for **__near.**

__linear - the __linear keyword is used as a pointer qualifier when accessing data in the extended (higher than 64KiB) memory on MMU-enabled devices.

LINEAR - the linear keyword is used to specify linear addresses in extended (higher than 64KiB) memory on MMU-enabled devices. For example:

```
// declare a variable at linear address 0x010000
const char my_var @ LINEAR 0x010000 = 10;
```

__alignof__ - returns the alignment of the specified type.

__va_sizeof__ - returns the size (in bytes) of the specified data type (even after an eventual automatic casting).

__interrupt - this keyword tells the compiler that the function is called automatically by the hardware when an interrupt occurs. The keyword must be followed by a numeric constant which specifies the interrupt vector number. The compiler also supports the **interrupt** alias for **__interrupt**. Since version 6.0, the Codewarrior IDE includes pre-defined symbols for all interrupt vectors (refer to tables 2.1 to 2.6 for all related symbols):

```
// SWI interrupt servicing function
void interrupt VectorNumber_Vswi void swi_service(void);
// IRQ interrupt servicing function
void interrupt VectorNumber_Virq trata_irq(void);
// ADC interrupt servicing function
void interrupt VectorNumber_Vadc trata_adc(void);
```

__asm - allows assembly instruction insertion into the C source code. We will discuss this operation in greater details later in this chapter. The compiler also supports the **asm** alias for **__asm**. Example:

```
// Insert a STOP instruction
asm stop;
```

Page 127 and 128: Text related to CODE_SEG, CONST_SEG, DATA_SEG should be readed as follows (also added an entry for STRING_SEG):

CODE_SEG – to specify a memory segment in which the functions will be stored. Example:

```
#pragma CODE_SEG MY_SEG
// functions following the pragma are all stored at MY_SEG segment
void my_func (void)
{
   ...
}

#pragma CODE_SEG DEFAULT
// functions following the pragma are all stored at the
// default segment (usually DEFAULT_ROM)
```

You can only use the segment names specified in the PLACEMENT section on the linker configuration file (.prm)!

CONST_SEG – to specify the memory segment in which the constant data will be stored. Example:

```
#pragma CONST_SEG MY_CONST_SEG
// const variables following the pragma are stored into
// MY_CONST_SEG memory segment
const char myconst1=50;

#pragma CONST_SEG DEFAULT
// const variables following the pragma are stored into
// the default const segment (usually ROM_VAR)
```

You can only use the segment names specified in the PLACEMENT section on the linker configuration file (.prm)!

DATA_SEG – this option works like the CONST_SEG but is valid for variables. All options valid for CONST_SEG are also valid for DATA_SEG. By default, data is stored into the DEFAULT_RAM segment.

STRING_SEG – this option works like the CONST_SEG but is valid for strings. All options valid for CONST_SEG are also valid for STRING_SEG. By default, strings are stored into the STRINGS segment.

Page 129: Remove entries for #pragma INTO_ROM and #pragma profile as they do not apply to the HCS08.

Pages 130 and 131: Include non-qualified pointers in the text.

- **Small**: this is the default model. All pointers and functions are addressed using 16-bit addresses, allowing code and data to be located anywhere within the 65536 possible addresses. It is possible to place variables into the direct page (by using #pragma DATA_SEG directive) and use the **__near** modifier to instruct the compiler to use the DIR addressing mode (which is faster than the default EXT addressing mode used in the small memory model). This model uses one of the following standard ANSI libraries:
- **Tiny**: on this model, all variables and the stack are placed into the direct page (first 256 memory addresses). Non-qualified pointers use 8-bit addressing mode (DIR). This memory model is faster than the small model but with a limited memory space (it is still possible to use the **__far** keyword to place variables outside the direct page). This model uses one of the following standard ANSI libraries:

Page 134 and 135: Modify asm occurrences to __asm:

- Global variables are stored at absolute addresses, whereas local variables and function formal parameters are stored in the stack. Accessing C variables within the assembly code can be done by simply writing the name of the variable as the assembly instruction operand. All addresses are assumed to be **char***. For example:

```
__asm lda aux      // load A with the value of the variable "aux"
__asm sta result   // store A in result
```

```
void store_at_pointer (char *my_pointer, char newvalue)
{
  __asm
  {
    lda     newvalue
    ldhx    my_pointer
    sta     ,x
  }
}
```

Listing 4.4

```
char my_asm_divide (char v1, char v2)
{
  __asm
  {
    lda   v1
    ldx   v2
    clrh
    div
  }
}
```

Listing 4.5

Page 139: Modify asm occurrences to __asm:

```
void reset(void)
{
  unsigned int temp;
  temp = 0x9E00;
  __asm
  {
    LDHX  @temp       // load the address of temp into H:X
    JMP   ,X          // jump to the address pointed to by H:X
  }
}
```

Listing 5.1

Page 140: Modify asm occurrence to __asm:

```
...
void reset(void)
{
  unsigned int temp = 0x9E00;
  __asm
  {
    LDHX  @temp      // load the address of temp into H:X
    JMP   ,X         // jump to the address pointed to by H:X
  }
}
...
```

Example 5.1

Page 156 (1st edition) or 158 (2nd edition): Topic 6.1.1.1, replace the or operator | by the and operator &:

```
PTAD = PTAD & ~0x01;       // clear bit 0 of PTA (clear pin PTA0)
```

Or:

```
PTAD = PTAD & ~BIT_0;      // clear bit 0 of PTA (clear pin PTA0)
```

Page 157: Topic 6.1.1.2, replace the or operator | by the and operator &:

```
pin_state = PTAD & 0x04;
```

Or:

```
pin_state = PTAD & BIT_2;
```

Page 259 and 260: Example 9.15 should be modified as follows:

```
...
#pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE
struct
{
  char tx_enable    : 1;             // transmitter is enabled
  char rx_flag      : 1;             // character received
} __near flags;
#pragma DATA_SEG DEFAULT

// This is the timer isr that generates the transmittion timing
void interrupt VectorNumber_Vtpmch0 tpmch0_isr(void)
{
  static char tx_state;
  #pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE
```

This places the next variable into the direct page for faster access times!

```
  static  __near char buffer;
  #pragma DATA_SEG DEFAULT
  TPMC0SC_CH0F = 0;                  // clear interrupt flag
  TPMC0V += BITTIME;                 // next compare in 104us
...

// This is the timer isr that generates the receiver timing
void interrupt VectorNumber_Vtpmch1 tpmch1_isr(void)
{
  static char rx_state;
  #pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE
```

This places the next variable into the direct page for faster access times!

```
  static  __near char buffer;
  #pragma DATA_SEG DEFAULT
  TPMC1SC_CH1F = 0;                  // clear interrupt flag
  TPMC1V += BITTIME;                 // next compare in 104us
...
```

Example 9.15 – Bit-banged TX/RX

Pages 274 and 275 should read as follows:

The internal temperature sensor outputs a DC voltage proportional to the chip internal temperature (typically 701.2mV at 25ºC for 1.8-3.6V devices or 1.396V for 5V devices). **The manufacturer advises using long sampling mode and a maximum of 1 MHz ADCK.**

The current temperature can be calculated by using the following formula:

$$\text{Temp}(^{o}\text{C}) = 25 - \frac{V_{sensor} - V_{25degrees}}{m}$$

In which: V_{sensor} is the current voltage output of the temperature sensor.

$V_{25degrees}$ is the output voltage when the sensor is at 25ºC (typically 701.2mV or 1.396V depending on the device).

m is the temperature slope (for 1.8 – 3.6V devices, m=1.646mV/ºC for temperatures among -40 and +25ºC, m=1.769mV/ºC for temperatures among +25 and +85ºC, for 5V devices, m=3.266mV/ºC for temperatures among -40 and +25ºC, m=3.638mV/ºC for temperatures among +25 and +85ºC).

Page 325: Example 11.10 should read as follows:

```
...
#pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE
struct
{
  char rx_flag      : 1;     // there are characters in the rx buffer
  char tx_buf_empty : 1;     // the tx buffer is empty
} __near flags;
#pragma DATA_SEG DEFAULT
...
```

Example 11.10

Page 327: Example 11.10 should read as follows:

```
// Function for reading a character from the rx buffer
char read_char_sci_buffer(void)
{
  char temp;
  temp = rxbuffer[rx_buf_read_pointer++];
  // if the read pointer is bigger than the buffer size, set the pointer to zero
  if (rx_buf_read_pointer>=RX_BUF_SIZE) rx_buf_read_pointer = 0;
  // if the read pointer reached the write pointer, clear the rx_flag
  SCI1C2_RIE = 0;          // disable the receive interrupt
  if (rx_buf_read_pointer==rx_buf_write_pointer) flags.rx_flag = 0;
  SCI1C2_RIE = 1;          // re-enable the receive interrupt
  return (temp);           // return the character
}

void main(void)
{
  char rxchar;
  SOPT1 = bBKGDPE;              // enable the debug pin
  PTCDD = BIT_0 | BIT_5;       // PTC0 and PTC5 as outputs
  // Following a reset, BUSCLK = 4MHz
  SCI1BD = 26;                  // SCI baud rate = 4MHz/(16*26)=9615 bps
  // enable TX and RX section, RX and TX interrupts
  SCI1C2 = bTIE | bRIE | bTE | bRE;
  PTCD_PTCD5 = 1;              // enable MAX3218
  ...
```

A more efficient approach is the use of a logical AND operation to mask the read pointer (this is valid only for power-of-2 buffer sizes): rx_buf_read_pointer &= (RX_BUF_SIZE-1);

Disabling and re-enabling the RIE flag prevents that the **if** statement disables the rx_flag if a new character is received during the executing of the statement!

Example 11.10

Page 354: Figure 13.3 should be changed to include the BDM connections:

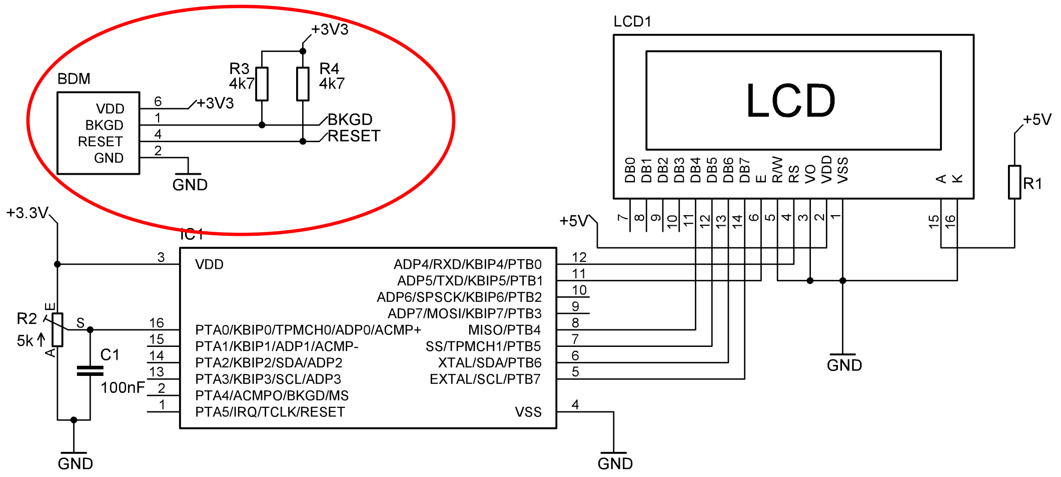

Figure 13.3 – Alphanumeric LCD wiring

Page 361: BDM wiring should be changed as follows:

Figure 13.4 – Serial servo controller

Page 362: Example 13.3 should be modified as follows:

```
...
#pragma DATA_SEG __DIRECT_SEG MY_ZEROPAGE
struct
{
  char rx_flag      : 1;     // new data received
  char seq_complete : 1;     // servo sequence is completed
} __near flags;
#pragma DATA_SEG DEFAULT
...
```

Example 13.3 – Serial servo controller

Page 365: The following figure is missing:

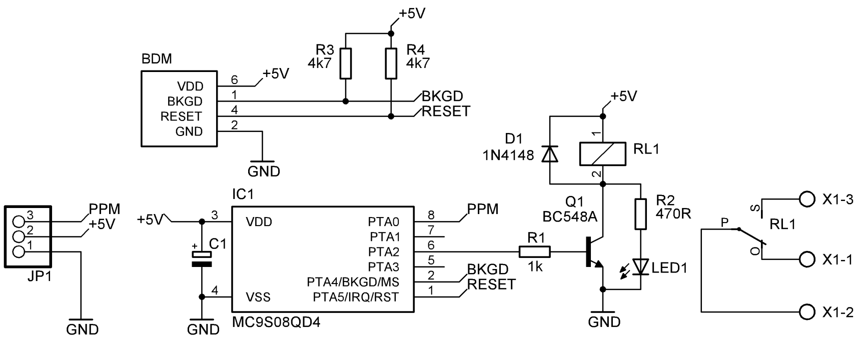

Figure 13.5 – PPM relay controller

Page 367: figure 13.6 (now 13.7) should be changed as follows:

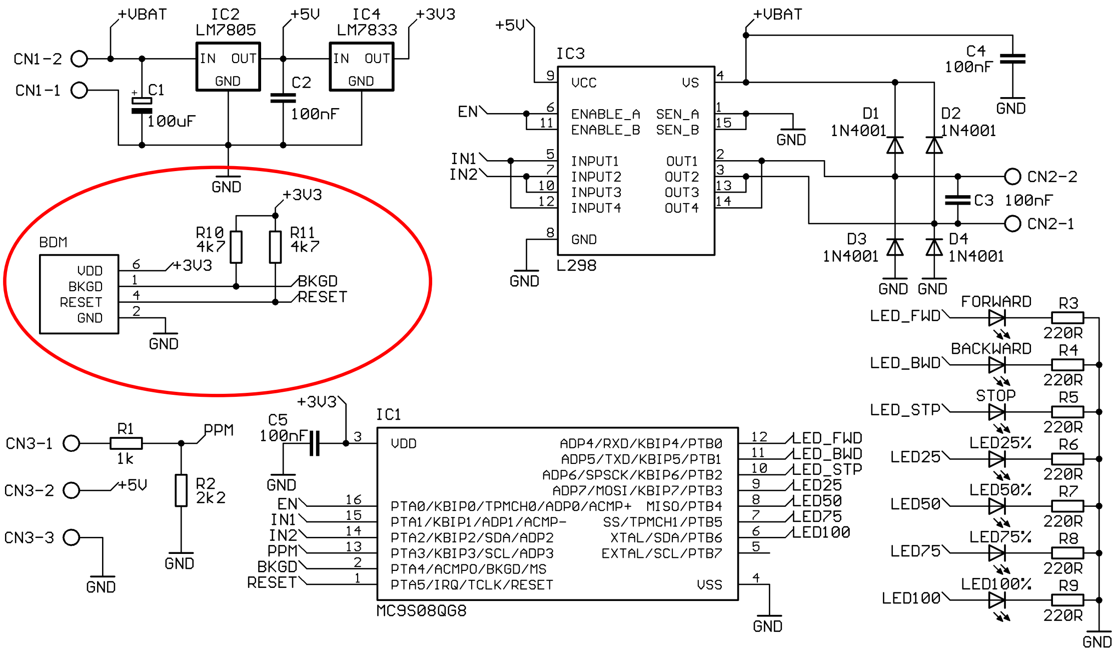

Figure 13.7 – DC motor speed controller

Page 371: figure 13.7 (now 13.8) should be changed as follows:

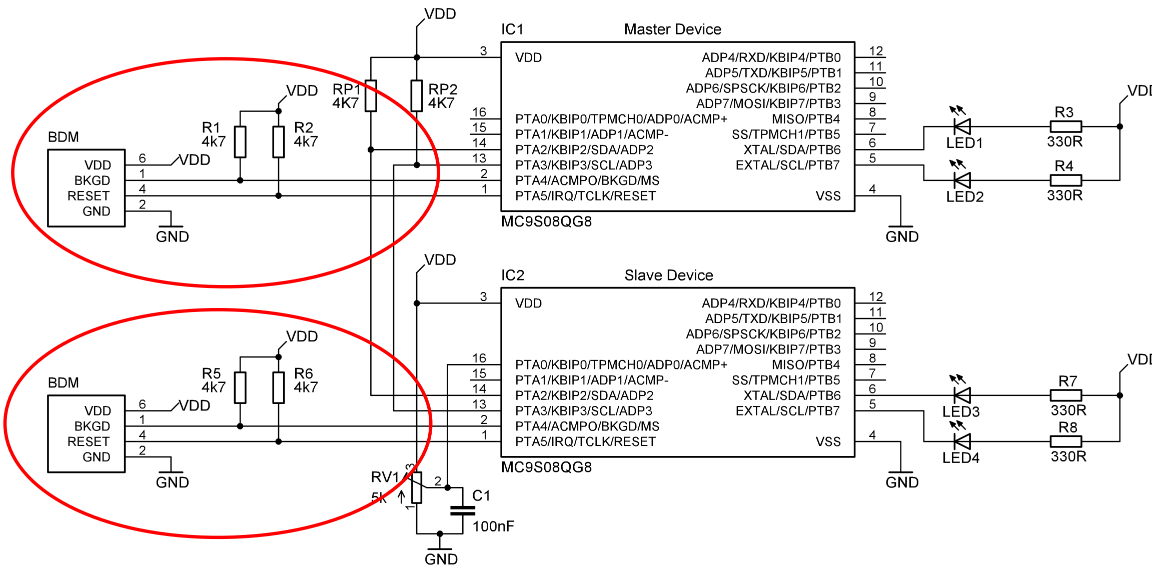

Figure 13.8 – Master/slave I^2C ADC and I/O expander (V_{DD}=3.3V)

Page 374: the last paragraph should read as follows:

The conversion result read from the slave device is also compared to a fixed threshold (511). If it is lower than 511, LED2 (connected to the master PTB7 pin) lights. If the value read is higher than or equal to 511, LED2 is turned off.

Page 382: example 13.8 should be modified as shown:

```
...
    // look for an octave postfix
    if (*song>='0' && *song<='9')
    {
      temp_octave = *song - '0';// the temporary octave is set accordingly
      song++;                  // advance to the next character
    }
    if (*song=='.') // a dot can also be found after the octave (???)
    {
      dot_flag = 1;            // if a '.' is found, set the flag
      song++;                  // advance to the next character
    }
    while (*song == ',') song++;    // skip the ','
    // calculate the note duration
    calc_duration = (60000/tempo)/(temp_duration);
    calc_duration *= 4;          // a whole note has four beats
    // check if the dot flag is set, if it is set, extend the duration in 50%
    if (dot_flag) calc_duration = (calc_duration*3)/2;
...
```

A dotted note multiplies the note duration by 3/2 (1.5 times)

Example 13.8 – RTTTL music player

Page 386 through 391: the last part of the play() and main() functions should be modified as shown:

```
...
    // calculate the note duration
    calc_duration = (60000/tempo)/(temp_duration);
    calc_duration *= 4;         // a whole note has four beats
    // check if the dot flag is set, if it is set, extend the duration in 50%
    if (dot_flag) calc_duration = (calc_duration*3)/2;
...

void main(void)
{
  unsigned char song_sel;
  SOPT1 = bBKGDPE;                   // enable the debug pin
  ICSSC = DCO_MID | NVFTRIM;         // configure FTRIM value, select DCO high range
  ICSTRM = NVICSTRM;                 // configure TRIM value
  ICSC1 = ICS_FLL | bIREFS;          // select FEI mode (ICSOUT = DCOOUT = 1024 * IRCLK)
  ICSC2 = BDIV_1;                    // ICSOUT = DCOOUT / 1
  // BUSCLK = 16MHz
  TPM1SC = TPM_BUSCLK | TPM_DIV4;    // TPMCK = 4MHz
  // set channel 0 to compare mode (interrupt only), 1ms interrupts
  TPM1C0V = 3999;
  TPM1C0SC = bCHIE | TPM_COMPARE_INT;
  // enable long sampling, 12-bit mode, ADICLK = 01b, ADCK = BUSCLK/16
  // ADC sampling rate = 25ksps
  ADCCFG = bADLSMP | ADC_12BITS | ADC_BUSCLK_DIV2 | ADIV_8;
...
```

Example 13.10 – Music-shake