MOTOROLA
intelligence everywhere™

digital dna™

8-Bit Software Development Kit for Motor Control Algorithms Library

User's Guide

M68HC08
Microcontrollers

SDKALUG/D
Rev. 1, 11/2002

MOTOROLA.COM/SEMICONDUCTORS

8-Bit Software Development Kit for Motor Control Algorithms Library

User's Guide

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://motorola.com/semiconductors/

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Revision History

Date	Revision Level	Description	Page Number(s)
July, 2002	N/A	Original release	N/A
November, 2002	1	Added **Section 5. Dead-Time Distortion Correction Algorithm**	73

List of Sections

List of Sections

Table of Contents

Section 1. Basic Fractional Math Library

Section 2. Controllers

Section 3. Motor Control 3-Phase Wave Generation

Section 4. Volts-per-Hertz (V/Hz) Table

Section 5. Dead-Time Distortion Correction Algorithm

Table of Contents

List of Figures and Tables

List of Figures and Tables

Section 1. Basic Fractional Math Library

1.1 Contents

1.2 Introduction

The software development kit (SDK) basic fractional math library performs the basic math for 8-bit and 16-bit fractional values.

1.3 API Definition

This section defines the Application Programming Interface (API).

Required Files:

```
#include "types.h"
#include "sdkmath.h"
```

NOTE: *The included files must be kept in order.*

1.3.1 8-Bit Fractional Math Interface

```
SByte add_8(SByte x, SByte y)
SByte lim_8(SByte x, SByte limit)
SByte neg_8(SByte x)
SByte shl_8(SByte x, UByte n)
SByte sub_8(SByte x, SByte y)
SByte sdiv_8(SWord16 x, UByte y)
SWord16 smul_8(SByte x, UByte y)
LIM_S8(x,limit)
```

1.3.2 16-Bit Fractional Math Interface

```
SWord16 add(SWord16 x, SWord16 y)
SWord16 lim(SWord16 x, SWord16 limit)
SWord16 neg(SWord16 x)
SWord16 shl (SWord16 x, UByte n)
SWord16 sub(SWord16 x, SWord16 y)
LIM_S16(x,limit)
```

1.3.3 Other Function Math Interface

```
UWord16 umul_16x8(UWord16 x, UByte x)
SWord16 smul_16x8(SWord16 x, UByte y)
UByte udiv_16to8(UWord16 x, UWord16 y)
```

1.4 API Specification

The following specifies the Application Programming Interface (API).

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Implemented as description:

fc — function in C code

fa — function in assembly code

M — macro

Table 1-1. Mathematical Function Description

Function	Parameters			Return (out)	Description	Notes[1]
SWord16 add (SWord16 x, SWord16 y)	x y	in in	<-32768..32767> <-32768..32767>	<-32768..32767>	return saturated value (x+y)	fa
SByte add_8 (SByte x, SByte y)	x y	in in	<-128..127> <-128..127>	<-128..127>	return saturated value (x+y)	fa
SWord16 lim (SWord16 x, SWord16 limit)	x limit	in in	<-32768..32767> <0..32767>	<-limit..limit>	return limited value	fc
SByte lim_8 (SByte x, SByte limit)	x limit	in in	<-128..127> <0..127>	<-limit..limit>	return limited value	fc
SWord16 neg(SWord16 x)	x	in	<-32768..32767>	<-32768..32767>	return saturated value -x)	fa
SByte neg_8(SByte x)	x	in	<-128..127>	<-128..127>	return saturated value (-x)	fa
SWord16 shl (SWord16 x, UByte n)	x n	in in	<-32768..32767> <0..16>	<-32768..32767>	return saturated value (x*2^n)	fa
SByte shl_8 (SByte x, UByte n)	x n	in in	<-128..127> <0..8>	<-128..127>	return saturated value (x*2^n)	fa
SWord16 sub (SWord16 x, SWord16 y)	x y	in in	<-32768..32767> <-32768..32767>	<-32768..32767>	return saturated value (x-y)	fa
SByte sub_8(SByte x, SByte y)	x y	in in	<-128..127> <-128..127>	<-128..127>	return saturated value (x-y)	fa
UByte udiv_16to8 (UWord16 x, UWord16 y)	x y	in in	<0..65535> <0..65535>	<0..255>	return 256*x/y	fa
UWord16 umul_16x8 (UWord16 x, UByte y)	x y	in in	<0..65535> <0..255>	<0..65535>	return x(L)*y/256+x(H)*y)	fc
SWord16 smul_16x8 (SWord16 x, UByte y)	x y	in in	<-32768..32767> <0..255>	<-32768..32767>	return x(L)*y/256+x(H)*y)	fc
SWord16 smul_8 (SByte x, UByte y)	x y	in in	<-128..127> <0..255>	<-32385..32385>	return x*y	fc
SByte sdiv_8 (SWord16 x, UByte y)	x y	in in	<-32768..32767> <0..255>	<-128..127>	return x*y	fc
SWord16 LIM (SWord16 x, SWord16 limit)	x limit	in in	<-32768..32767> <0..32767>	<-limit..limit>	return limited value	M
SByte LIM_8 (SByte x, SByte limit)	x y	in in	<-128..127> <0..127>	<-limit..limit>	return limited value	M

1. fc — function in C code
   fa — function in assembly code
   M — macro

Table 1-2. Memory Consumption an Execution Time

Function Name	Size (Bytes)	Clock Cycles[1]		
		Minimum	Typical[2]	Maximum
add	25	58	58	62
add_8	16	34	34	36
lim	64	76	79	94
lim_8	31	45	45	53
neg	12	29	29	38
neg_8	7	20	23	24
shl	46	53	65+n*11	233
shl_8	24	24	31+n*7	86
sub	30	68	68	72
sub_8	16	34	34	36
udiv_16to8	35	44	150	338
umul_16x8	26	76	76	76
smul_16x8	54	83	137	146
smul_8	36	42	87	91
sdiv_8	52	73	85	94

1. The execution time includes both parameters passing and function calls. This is true because the execution time can be different according to change of parameters passing time.
2. The typical clock cycles represent time for common parameter values. These are typically parameters which do not cause any saturation.

1.5 Function Description

1.5.1 add

Call(s):

```
SByte add_8    (SByte x, SByte y);
SWord16 add    (SWord16 x, SWord16 y);
```

Description:

The add function returns addition of two fractional numbers *result = x + y.* The result is limited to the maximum or minimum fractional values.

Range Issues: None

Example:

```
SByte          x8, y8, z8;
SWord16        x16, y16, z16;

x8 = 100; y = 30;
x16 = 3400; y = -5200;

z8 = add_8(x8, y8);
z16 = add(x16, y16);
```

Result:

```
z8: 127
z16: -1800
```

1.5.2 lim

Call(s):

```
SByte lim_8    (SByte x, SByte limit);
SWord16 lim    (SWord16 x, SWord16 limit);
```

Description:

The lim function returns values limited in range specified by limit.

Range Issues:

```
limit:          <0..127> (lim_8)
                <0..32767> (lim)
```

Example:

```
SByte           x8, lim8, z8;
SWord16         x16, lim16, z16;

x8 = 115; lim8= 100;
x16 = -2456; lim16 = 1000;

z8 = lim_8(x8, y8);
z16 = lim(x16, y16);
```

Result:

```
z8: 100
z16: -1000
```

1.5.3 neg

Call(s):

```
SByte neg_8    (SByte x);
SWord16 neg    (SWord16 x)
```

Description:

The sub function returns the negation of a fractional input *result* = –*x*. The result is limited to the maximum or minimum fractional values.

Range Issues: None

Example:

```
SByte          x8, z8;
SWord16        x16, z16;

x8 = -128;
x16 = 12500;

z8 = neg_8(x8);
z16 = neg(x16);
```

Result:

```
z8: 127
z16: -12500
```

1.5.4 shl

Call(s):

```
SByte shl_8    (SByte x, UByte n);
SWord16 shl    (SWord16 x, UByte n);
```

Description:

The sub function returns the value x arithmetical shifted by n bits *result = x * 2^n*. The result is limited to the maximum or minimum fractional values.

Range Issues:

```
n :            <0..7> (shl_8)
               <0..15> (shl)
```

Example:

```
SByte          x8, z8;
SWord16        x16 z16;
UByte          n;

x8 = -108;
x16 = 7000;
n = 2;

z8 = shl_8(x8, n);
z16 = shl(x16, n);
```

Result:

```
z8: -128
z16: 28000
```

1.5.5 sub

Call(s):

```
SByte sub_8    (SByte x, SByte y);
SWord16 sub    (SWord16 x, SWord16 y);
```

Description:

The sub function returns the subtraction of two fractional numbers *result* = *x* – *y*. The result is limited to the maximum or minimum fractional values.

Range Issues: None

Example:

```
SByte          x8, y8, z8;
SWord16        x16, y16, z16;

x8 = 100; y8 = 30;
x16 = 25400; y16 = -9200;

z8 = sub_8(x8, y8);
z16 = sub(x16, y16);
```

Result:

```
z8: 70
z16: 32767
```

1.5.6 udiv_16to8

Call(s):

```
UByte6 udiv_16to8    (UWord16 x, UWord16 y);
```

Description:

Unsigned dividing 16 bit x by high 8 bit from 16 bit y, *result = 256*x/y.* Both divisor and dividend are scaled to get high result precision. The result is saturated at 256 if overflow occurs.

Range Issues:

if y=0 the result is saturated at 0xFF

Example:

```
UByte          z8;
UWord16        x16, y16;

x16=3426 /* 1101 01100010b */
y16 =14835 /* 111001 11110011 */

z8 = udiv_16_to_8(x16, y16);
```

Result:

```
z8: 59 /* 13704 (110101 10001000b) / 231 (11100111b) */
```

1.5.7 umul_16x8

Call(s):

```
UWord16 umul_16x8    (UWord16 x, UByte y);
```

Description:

Unsigned multiply 16 bit x by 8 bit y, *result = x(L) * y/256 + x(H) * y.*

Simplified Description:

return x * y / 256

Range Issues: None

Example:

```
UByte          y8;
UWord16        x16, z16;

x16=3426
y8 =148

z16 = umul_16x8(x16, y8);
```

Result:

```
z16: 1980
```

1.5.8 smul_16x8

Call(s):

```
SWord16 smul_16x8    (SWord16 x, UByte y);
```

Description:

Signed multiply 16 bit signed x by 8 bit unsigned y,
*result = x(L) * y/256 + x(H) * y.*

Simplified Description:

return x * y / 256

Range Issues: None

Special Issues:

This function uses the absolute value of x. If x = –32768, the value is limited at –32767. For that reason, the function can return unexpected results (e.g., –6399 instead of –6400 for x = –32768 and y = 50).

Example:

```
UByte          y8;
SWord16        x16, z16;

x16=-3426
y8 =148

z16 = smul_16x8(x16, y8);
```

Result:

```
z16: -1980
```

1.5.9 smul_8

Call(s):

```
SWord16 smul_8 (SByte x, UByte y);
```

Description:

Signed multiply 8 bit signed x by 8 bit unsigned y, *result = x * y*

Range Issues: None

Special Issues:

This function uses the absolute value of x. If the x = –128, the value is limited at –127. For that reason, the function can return unexpected result (e.g., –6350 instead of –6400 for x = –128 and y = 50).

Example:

```
UByte           y8;
SByte           x8;
SWord16         z16;

x8= -100
y8 = 80

z16 = smul_8(x8, y8);
```

Result:

```
z16: -8000
```

1.5.10 sdiv_8

Call(s):

```
SByte sdiv_8   (SWord16 x, UByte y);
```

Description:

Signed division 16 bit signed x by 8 bit unsigned y, *result = x / y*

Range Issues:

If y = 0 and x is positive, the result is saturated at 127. If y = 0 and x is negative the result is saturated at -128.

Example:

```
UByte          y8;
SWord16        x16;
SByte          z8;

x16= -8000
y8 = 80

z8 = sdiv_8(x8, y8);
```

Result:

```
z8: -100
```

1.6 Macro Description

1.6.1 LIM

Call(s):

```
SByte LIM_8    (SByte x, SByte limit);
SWord16 LIM    (SWord16 x, SWord16 limit);
```

Description:

The lim function returns value limited in range specified by limit.

Range Issues:

```
limit:          <0..127> (LIM_8)
                <0..32767> (LIM)
```

Example:

```
SByte           x8, lim8, z8;
SWord16         x16, lim16, z16;

x8 = 115; lim8= 100;
x16 = -2456; lim16 = 1000;

z8 = LIM_8(x8, y8);
z16 = LIM(x16, y16);
```

Result:

```
z8: 100
z16: -1000
```

1.7 Trigonometric Math Functions

This subsection describes the trigonometric functions, which are included in 8-bit SDK.

1.7.1 API Definition

This section defines the Application Programming Interface (API).

Required files:

```
“sin.asm”
“sinlut.asm”


Required includes:
#include "types.h"
#include "sin.h"
```

NOTE: *The include files must be kept in this order.*

1.7.2 sinPIxLUT

Public Interface:

```
SWord16 sinPIxLUT(UByte amplitude, SWord16 phase);
```

Description:

This function calculates the sine value of an argument phase multiplied by amplitude utilizing the 256 x 8 bit look-up table.

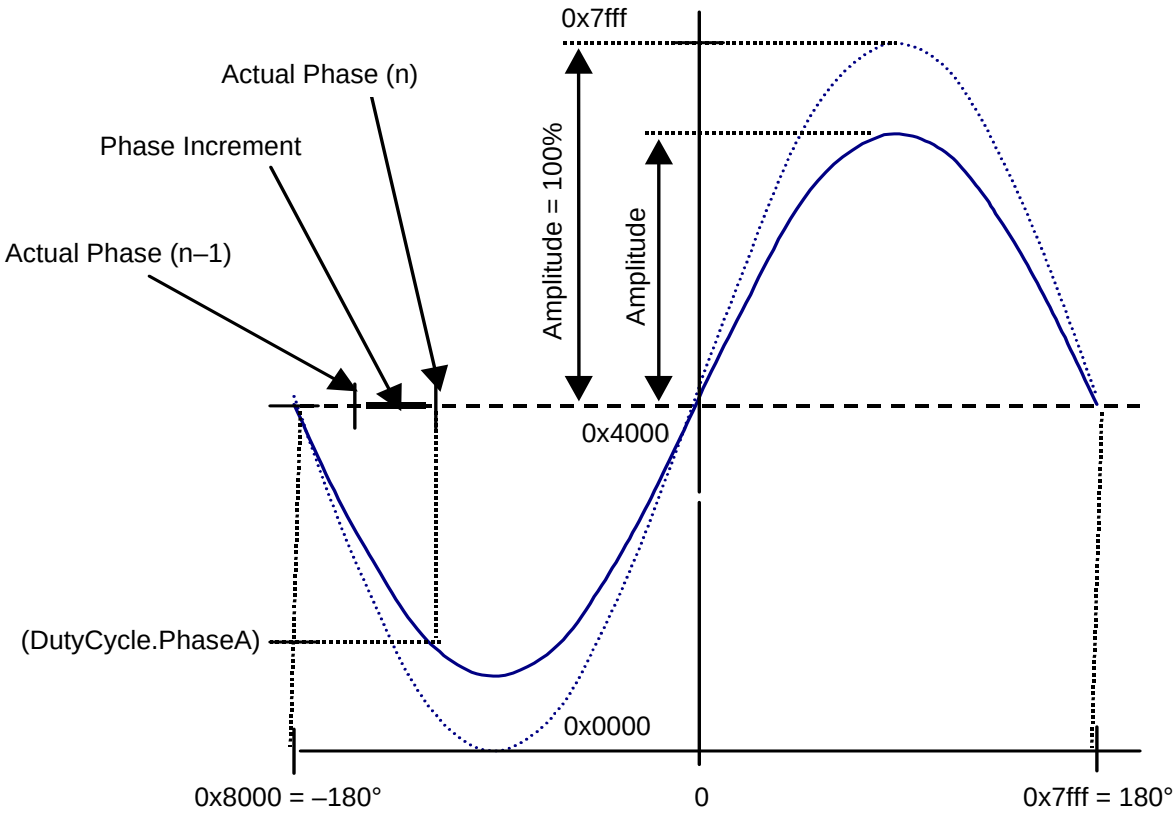


Figure 1-1. Since Wave Generation

1.7.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

1.7.4 sinPIxLUT

Call(s):

```
SWord16 sinPIxLUT(UByte amplitude, SWord16 phase);
```

Table 1-3. sinPIxLUT Parameters

Variable	Direction	Range	Explanation
return	out	–0x8000..0x7FFF	Wave output value (–1..1)
amplitude	in	0..0xFF	Desired phase voltage amplitude
phase	in	0..0xFFFF	Actual phase

Description:

The *sinPIxLUT* function from given *amplitude* and *phase* calculates an immediate value of the sinus. The shape of the generated waveforms depends on the data stored in the sine table. In motor control applications data usually describes the pure sinewave of sinewave with addition of third harmonic component.

Returns:

Sine value multiplied by the required amplitude for the actual phase.

Range Issues:

To ensure proper wave generation, arguments must be within the following limits:

amplitude must be within the fractional range:

0x00 <= *amplitude* <= 0xFF for 0%..100% of generated sine amplitude

0x8000 <= *phase* <= 0x7FF for –100%..100% of sine period (0x8000..0x7FFF)

Execution Time:

Minimum of 75 cycles and maximum 81 cycles including function call

Memory Consumption:

27 bytes

Special Issues:

The result is saturated.

Example:

```
UByte           amplitude;
SWord16         phase,sineoutput;


amplitude = 115;
phase = -2456;

sineoutput = sinPIxLUT(amplitude, phase);
```

Result:

```
sineoutput: -3392
```

Section 2. Controllers

2.1 Contents

2.2 Introduction

This section describes the API for standard controllers (e.g., PI and PID) for use in motor control applications in general.

The controller algorithms are used to control motor speed, current, shaft position, etc. operating in closed loop.

2.3 API Definition

This section defines the API for general controllers.

The header files *controller.h* and *types.h* include all required prototypes and structure/type definitions.

Public Interface Function(s):

```
SWord16 controllerPI_8(SByte desiredValue, SByte measuredValue,
sPIparams *pParams)

SWord16 controllerPI_Scl_8(UByte scale, SByte desiredValue, SByte
measuredValue, sPIparams *pParams)

SWord16 controllerPI_Lim_8(SByte desiredValue, SByte
measuredValue, sPIparams *pParams, SWord16 NegativePILimit,
SWord16 PositivePILimit)

SWord16 controllerPI(SWord16 desiredValue, SWord16 measuredValue,
sPIparams *pParams)
```

Controller Function Types:

See **Table 2-1**.

Table 2-1. Controller Function Types

`controllerPI_8( )`	The function calculates the standard PI (proportional-integral) controller. The integral calculation is approximated by using the Backward Euler method, also known as Backward Rectangular or right-hand approximation. All variables are saturated.
`controllerPI_Scl_8( )`	The function calculates the standard PI controller. The integral calculation is approximated by using the Backward Euler method, also known as Backward Rectangular or right-hand approximation. All variables are saturated. The control error is scaled by 2^{scale}.
`controllerPI_Lim_8( )`	The function calculates the standard PI controller. The integral calculation is approximated by using Backward Euler method, also known as Backward Rectangular or right-hand approximation. The limitations are independently used for integral portion and controller output and the limits can be set differently for positive and negative limitation.
`controllerPI( )`	The function calculates the standard PI controller. The integral calculation is approximated by using Backward Euler method, also known as Backward Rectangular or right-hand approximation. All variables are saturated. Both measured and desired value are SWord16.

Public Data Structure(s):

```
typedef struct
{
    UByte   ProportionalGain;
    UByte   IntegralGain;
    SWord16 IntegralPortionK_1;
}sPIparams;
```

Members:

See **Table 2-2**.

Table 2-2. *sPIparams* Data Structure Members

ProportionalGain	UByte	The scaled gain of the proportional controller portion
IntegralGain	UByte	The scaled gain of the integral controller portion
IntegralPortionK_1	SWord16	The integral portion in k-1 step

2.4 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

See **Table 2-3**.

Table 2-3. Memory Consumption and Execution Time

Function Name	SIZE (Bytes)	Clock Cycles		
		Minimum	Typical	Maximum
controllerPI_8	168	286	309	341
controllerPI_Scl_8	174	309	353	421
controllerPI_Lim_8	273	382	416	466
controllerPI	221	449	471	511

2.4.1 controllerPI_8

This function offers one method to calculate the PI algorithm.

Call(s):

```
SWord16 controllerPI_8(SByte desiredValue,
                       SByte measuredValue,
                       sPIparams *pParams)
```

Arguments:

See **Table 2-4**.

Table 2-4. *controllerPI_8* Arguments

desiredValue	in	Desired value
measuredValue	in	Measured value
pParams	inout	Pointer to variable containing controller parameters and the integral portion in k-1 step

Description:

The *controllerPI_8* function calculates the PI algorithm according to the following equations:

The PI algorithm in continuous time domain:

$$u(t) = K_c\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \qquad \textbf{Equation: 1}$$

The transfer function is shown below:

$$F(p) = K_c \cdot \left[1 + \frac{1}{T_I} \cdot \frac{1}{p}\right] = \frac{u(p)}{e(p)} \qquad \textbf{Equation: 2}$$

The PI algorithm in discrete time domain:

$$u(k) = K_c \cdot e(k) + u_I(k-1) + K_c \cdot \frac{T}{T_I} \cdot e(k) \qquad \textbf{Equation: 3}$$
$$e(k) = w(k) - m(k)$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right - hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$u(k) = u_P(k) + u_I(k)$$

$$u_P(k) = K_c \cdot e(k)$$

$$u_I(k) = u_I(k-1) + K_c\frac{T}{T_I} \cdot e(k)$$

where:

e(k)	=	input error in step k
w(k)	=	desired value in step k
m(k)	=	measured value in step k
u(k)	=	controller output in step k
$u_p(k)$	=	proportional output portion in step k
$u_I(k)$	=	integral output portion in step k
$u_I(k-1)$	=	integral output portion in step k-1
T_I	=	integral time constant
T	=	sampling time
K_c	=	controller gain
t	=	time
p	=	Laplace variable

Returns:

The *controllerPI_8* function returns the SWord16 value representing the controller output in step k.

Range Issues: None

Special Issues:

Set proper value to *IntegralPortionK_1* before first calling is recommended.

Example 1. controllerPI_8

```
#include "types.h"
#include "controller.h"

void main(void)
{
          sPIparams    piParams;
          SByte        desiredValue, measuredValue;
          SWord16      piOutput;

          piParams.ProportionalGain      = 34;
          piParams.IntegralGain          = 25;
          piParams.IntegralPortionK_1    = 0;

          desiredValue  = 84;
          measuredValue = 115;

          piOutput = controllerPI_8(desiredValue, measuredValue, &piParams);
}

result -  PIoutput: -1829 [(84 - 115) * 25 + (84 - 115) * 34]
          piParams.IntegralPortionK_1: -775 [(84 - 115) * 25]
```

2.4.2 controllerPI_Scl_8

This function offers one method to calculate the PI algorithm.

Call(s):

`SWord16 controllerPI_Scl_8`	`(UByte scale, SByte desiredValue, SByte measuredValue, sPIparams *pParams)`

Arguments:

See **Table 2-5**.

Table 2-5. *controllerPI_Scl_8* arguments

scale	in	ProportionalGain and IntegralGain scale
desiredValue	in	Desired valueProportionalGain and IntegralGain scale
measuredValue	in	Measured value
pParams	inout	Pointer to variable containing controller parameters and the integral portion in k-1 step

Description:

The *controllerPI_Scl_8* function calculates the PI algorithm according to the following equations:

The PI algorithm in continuous time domain:

$$u(t) = K_c\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \qquad \textbf{Equation: 4}$$

The transfer function is shown below:

$$F(p) = K_c \cdot \left[1 + \frac{1}{T_I} \cdot \frac{1}{p}\right] = \frac{u(p)}{e(p)} \qquad \textbf{Equation: 5}$$

The PI algorithm in discrete time domain:

$$u(k) = K_c \cdot e(k) + u_I(k-1) + K_c \cdot \frac{T}{T_I} \cdot e(k) \qquad \textbf{Equation: 6}$$

$$e(k) = (2^{scale} \cdot (w(k) - m(k)))$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right-hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$u(k) = u_P(k) + u_I(k)$$

$$u_P(k) = K_c \cdot e(k)$$

$$u_I(k) = u_I(k-1) + K_c \frac{T}{T_I} \cdot e(k)$$

where:

e(k)	=	input error in step k
w(k)	=	desired value in step k
m(k)	=	measured value in step k
u(k)	=	controller output in step k
$u_p(k)$	=	proportional output portion in step k
$u_I(k)$	=	integral output portion in step k
$u_I(k-1)$	=	integral output portion in step k-1
T_I	=	integral time constant
T	=	sampling time
K_c	=	controller gain
t	=	time
p	=	Laplace variable

Returns:

The *controllerPI_Scl_8* function returns the SWord16 value representing the controller output in step k.

Range Issues: None

Special Issues:

Set proper value to *IntegralPortionK_1* before first calling is recommended.

Example 2. controllerPI_Scl_8

```
#include "types.h"
#include "controller.h"

void main(void)
{
          sPIparams    piParams;
          UByte        scale;
          SByte        desiredValue, measuredValue;
          SWord16      piOutput;

          piParams.ProportionalGain      = 130;
          piParams.IntegralGain          = 25;
          piParams.IntegralPortionK_1    = 0;

          desiredValue  = 84;
          measuredValue = 115;
          scale = 2

          piOutput = controllerPI_Scl_8(scale, desiredValue, measuredValue, &piParams);
}
result -  PIoutput: -19220 [(84 - 115) * 4 * 25 + (84 - 115) * 4 * 34]
          piParams.IntegralPortionK_1: -3100 [(84 - 115) * 4 * 25]
```

2.4.3 controllerPI_Lim_8

This function offers one method to calculate the PI algorithm.

Call(s):

```
SWord16 controllerLimPI_Lim_8  (SByte desiredValue,
                                SByte MeasuredValue,
                                sPIparams *pParams,
                                SWord16 NegativePILimit,
                                SWord16 PositivePILimit)
```

Arguments:

See **Table 2-6**.

Table 2-6. *controllerPI_Lim_8* arguments

desiredValue	in	Desired value
measuredValue	in	Measured value
pParams	inout	Pointer to variable containing controller parameters and the integral portion in k-1 step
NegativePILimit	in	Negative limit for controller output and integral portion
PositivePILimit	in	Positive limit for controller output and integral portion

Description:

The *controllerPI_Lim_8* function calculates the PI algorithm according to the following equations:

The PI algorithm in continuous time domain:

$$u(t) = K_c\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \qquad \textbf{Equation: 7}$$

The transfer function is shown below:

$$F(p) = K_c \cdot \left[1 + \frac{1}{T_I} \cdot \frac{1}{p}\right] = \frac{u(p)}{e(p)} \qquad \textbf{Equation: 8}$$

The PI algorithm in discrete time domain:

$$u(k) = K_c \cdot e(k) + u_I(k-1) + K_c \cdot \frac{T}{T_I} \cdot e(k) \qquad \textbf{Equation: 9}$$

$$e(k) = w(k) - m(k)$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right-hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$u(k) = u_P(k) + u_I(k)$$

$$u_P(k) = K_c \cdot e(k)$$

$$u_I(k) = u_I(k-1) + K_c \frac{T}{T_I} \cdot e(k)$$

where:

e(k)	=	input error in step k
w(k)	=	desired value in step k
m(k)	=	measured value in step k
u(k)	=	controller output in step k
$u_p(k)$	=	proportional output portion in step k
$u_I(k)$	=	integral output portion in step k
$u_I(k-1)$	=	integral output portion in step k-1
T_I	=	integral time constant
T	=	sampling time
K_c	=	controller gain
t	=	time
p	=	Laplace variable

Returns:

The *controllerPI_Lim_8* function returns the SWord16 value representing the controller output in step k.

Range Issues:

Controller output u and integral portion u_I are limited

<NegativePILimit .. PositivePILimit>

Special Issues:

Setting proper value to *IntegralPortionK_1* before first calling is recommended.

Example 3. controllerPI_Lim_8

```
#include "types.h"
#include "controllers.h"

void main(void)
{
          #define      POSITIVE_LIMIT 1000
          #define      NEGATIVE_LIMIT -(POSITIVE_LIMIT)
          sPIparams    piParams;
          SByte        desiredValue, measuredValue;
          SWord16      piOutput;

          piParams.ProportionalGain      = 34;
          piParams.IntegralGain          = 25;
          piParams.IntegralPortionK_1    = 0;

          desiredValue  = 84;
          measuredValue = 115;

          piOutput = controllerPI_Lim_8(desiredValue, measuredValue, &piParams,
                                        NEGATIVE_LIMIT, POSITIVE_LIMIT);
}

result -  PIoutput: -1000 [(84 - 115) * 25 + (84 - 115) * 34]
          piParams.IntegralPortionK_1: -775 [(84 - 115) * 25]
```

2.4.4 controllerPI

This function offers one method to calculate the PI algorithm.

Call(s):

```
SWord16 controllerPI   (SWord16 desiredValue,
                        SWord16 measuredValue,
                        sPIparams *pParams
```

Arguments:

See **Table 2-7**.

Table 2-7. *controllerPI* arguments

desiredValue	in	Desired value
measuredValue	in	Measured value
pParams	inout	Pointer to variable containing controller parameters and the integral portion in k-1 step

Description:

The *controllerPI* function calculates the PI algorithm according to the following equations:

The PI algorithm in continuous time domain:

$$u(t) = K_c\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \qquad \textbf{Equation: 10}$$

The transfer function is shown below:

$$F(p) = K_c \cdot \left[1 + \frac{1}{T_I} \cdot \frac{1}{p}\right] = \frac{u(p)}{e(p)} \qquad \textbf{Equation: 11}$$

The PI algorithm in discrete time domain:

$$u(k) = K_c \cdot e(k) + u_I(k-1) + K_c \cdot \frac{T}{T_I} \cdot e(k) \qquad \textbf{Equation: 12}$$

$$e(k) = w(k) - m(k)$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right - hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$u(k) = u_P(k) + u_I(k)$$

$$u_P(k) = K_c \cdot e(k)$$

$$u_I(k) = u_I(k-1) + K_c \frac{T}{T_I} \cdot e(k)$$

where:

e(k)	=	input error in step k
w(k)	=	desired value in step k
m(k)	=	measured value in step k
u(k)	=	controller output in step k
$u_p(k)$	=	proportional output portion in step k
$u_I(k)$	=	integral output portion in step k
$u_I(k-1)$	=	integral output portion in step k-1
T_I	=	integral time constant
T	=	sampling time
K_c	=	controller gain
t	=	time
p	=	Laplace variable

Returns:

The *controllerPI* function returns the SWord16 value representing the controller output in step k.

Range Issues: None

Special Issues:

Setting proper value to *IntegralPortionK_1* before first calling is recommended.

Example 4. controllerPI

```
#include "types.h"
#include "controllers.h"

void main(void)
{
          sPIparams    piParams;
          SWord16      desiredValue, measuredValue, piOutput;

          piParams.ProportionalGain      = 34;
          piParams.IntegralGain          = 25;
          piParams.IntegralPortionK_1    = 0;

          desiredValue  = 6540;
          measuredValue = 6180;

          piOutput = controllerPI(desiredValue, measuredValue, &piParams);
}

result -  PIoutput: 82[((6540 - 6180) * 25)/256 + ((6540 - 6180) * 34)/256]
          piParams.IntegralPortionK_1: 35  [((6540 - 6180) * 25)/256]
```

Section 3. Motor Control 3-Phase Wave Generation

3.1 Contents

3.2 Introduction

This section describes the algorithm for 3-phase sine wave generation used for general motor control. The signal generated and controlled by this algorithm is shown in **Figure 3-1**.

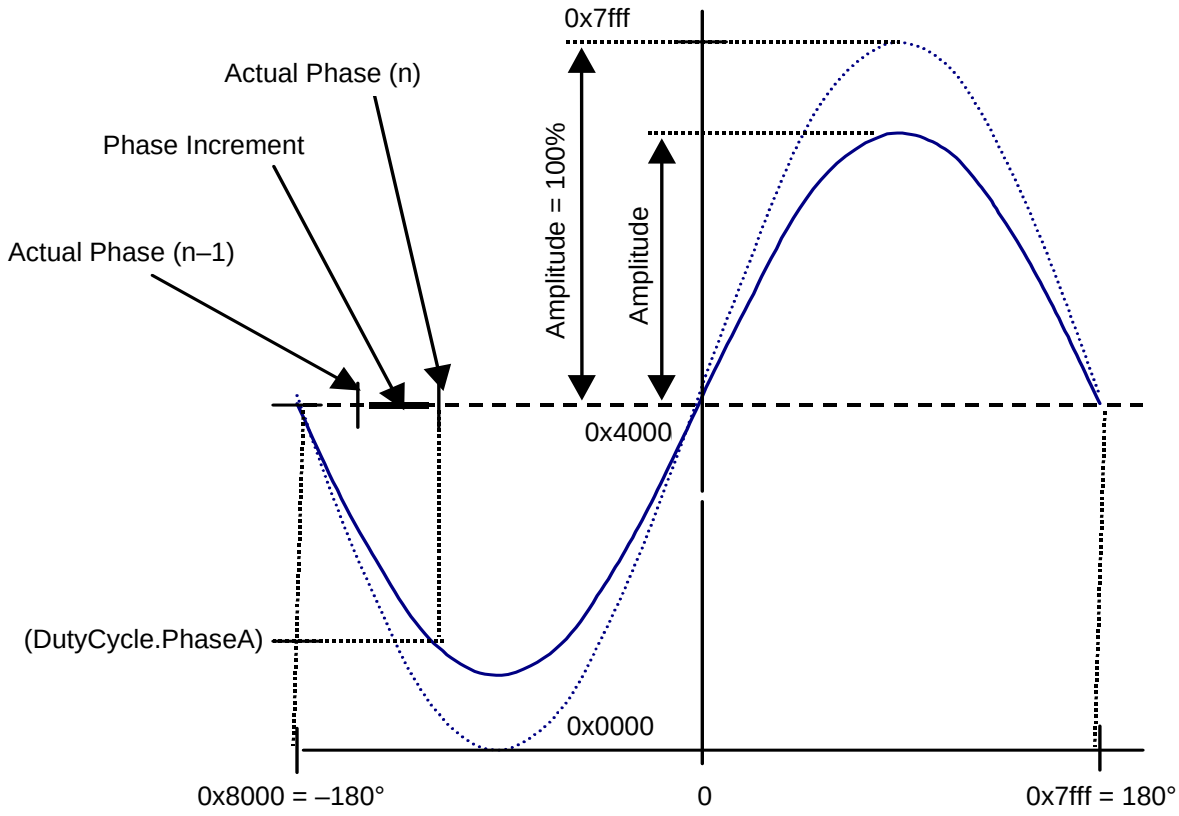


Figure 3-1. mcgen3PhWaveSine Data Explanation — Sine (Illustration only for Phase A. Phase B and C are shifted 120° with respect to Phase A)

3.3 API Definitions

This section defines the application programming interface (API). The header files *types.h* and *mcgen.h* include all required prototypes and structure/type definitions. This information is included here for the programmer's reference.

3.3.1 Public Interface Function(s)

```
/*****************************************************************************
*
*  Module:
*   UByte mcgenRippleCancel (UByte Amplitude, UByte u_dc_bus)
*
*  Description:
*   Function eliminates the influence of DC bus voltage ripples to generated
*   phase voltage waveforms.
*
*  Arguments: The function has two arguments:
*   in  - Amplitude - desired phase voltage amplitude
*   in  - u_dc_bus  - voltage in DC bus link
*
*  Returns:
*   UByte - corrected amplitude related to voltage in DC bus link
*
*  Range Issues: None
*
*  Special Issues: None
*
*****************************************************************************/

UByte mcgenRippleCancel (UByte Amplitude, UByte u_dc_bus);

*****************************************************************************
*
*  Module:
*   void mcgen3PhWaveSine(UByte Amplitude, SWord16 Phase, mc_s3PhaseSystem *pHandle)
*
*  Description:
*   Function calculates an immediate value of the three phase sine wave system
*   using a sine table. Individual waves are shifted 120 Deg. each other.
*
*   Phase =  Phase + PhaseInc
*
*   PWMA =   0.5 + Amplitude * sin(Phase)
*   PWMB =   0.5 + Amplitude * sin(Phase - 120 deg.)
*   PWMC = - 0.5 - PWMA - PWMB
*
*  Arguments: The function has three arguments:
*   in  - Amplitude - desired wave amplitude
*   in  - Phase     - desired phase of system
*   out - pHandle   - pointer to structure of mc_s3PhaseSystem type
*                     which contains the output data three phase system
*
*  Range Issues: None
*
*  Special Issues: None
*
*****************************************************************************/

void mcgen3PhWaveSine (UByte Amplitude, SWord16 Phase, mc_s3PhaseSystem * pHandle);
```

```
/*****************************************************************************
*
*  Module:
*   void mcgen3PhWaveSine3rdH(UByte Amplitude, SWord16 Phase,
*                          mc_s3PhaseSystem *pHandle)
*
*  Description:
*   Function calculates an immediate value of the three phase sine wave system
*   using a sine table. Individual waves are shifted 120 Deg. each other.
*
*   Phase =  Phase + PhaseInc
*
*   PWMA = 0.5 + Amplitude * 2/sqrt(3) * ((sin(Phase)           - 1/6 * sin(3*Phase))
*   PWMB = 0.5 + Amplitude * 2/sqrt(3) * ((sin(Phase - 120 deg) - 1/6 * sin(3*Phase))
*   PWMC = 0.5 + Amplitude * 2/sqrt(3) * ((sin(Phase + 120 deg) - 1/6 * sin(3*Phase))
*
*  Arguments: The function has three arguments:
*   in  - Amplitude - desired wave amplitude
*   in  - Phase     - desired phase of system
*   out - pHandle   - pointer to structure of mc_s3PhaseSystem type
*                     which contains the output data three phase system
*
*  Range Issues: None
*
*  Special Issues: None
*
*****************************************************************************/

void mcgen3PhWaveSine3rdH(UByte Amplitude, SWord16 Phase,
                       mc_s3PhaseSystem * pHandle);
```

3.3.2 Public Data Structure(s):

Data structure *mc_s3PhaseSystem* is defined in types.*h* header file. See **Table 3-1**.

```
typedef struct
{
                        SWord16 PhaseA;
                        SWord16 PhaseB;
                        SWord16 PhaseC;
} mc_s3PhaseSystem;
```

Table 3-1. *mc_s3PhaseSystem* Structure Elements

Variable	Explanation
PhaseA	Percentage of the wave output value for Phase A voltage (0%..100%)
PhaseB	Percentage of the wave output value for Phase B voltage (0%..100%)
PhaseC	Percentage of the wave output value for Phase C voltage (0%..100%)

3.4 API Specification

This section specifies the exact usege for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

3.4.1 *mcgenRippleCancel* — DC-Bus Ripple Cancellation Function

Call(s):

```
UByte mcgenRippleCancel (UByte Amplitude, UByte u_dc_bus);
```

Arguments:

See **Table 3-2**.

Table 3-2. *mcgenRippleCancel* Arguments

Amplitude	in	Desired phase voltage amplitude
u_dc_bus	in	Measured DC-bus voltage

Description:

The function *mcgenRippleCancel* converts the phase voltage amplitude (*Amplitude*) to the sine wave amplitude (*AmplitudeAmplScale*) based on the actual value of the DC-bus voltage (*u_dc_bus*). This eliminates the influence of DC-Bus voltage ripple to the generated phase voltage sine wave.

NOTE: *Both voltages must be in the same scale.*

The conversion coded in this function is described by the following formula:

AmplitudeAmplScale = *Amplitude* / (*u_dc_bus* / 2)

If the numerator (phase voltage amplitude) is greater than or equal to the denominator (half of the DC-bus voltage), the function returns sine wave amplitude equal to the maximum UByte value (the generated phase voltage sine wave amplitude is the maximum available, but still limited by the level of the DC-bus voltage).

Returns:

UByte value of the sine wave amplitude (*AmplitudeAmplScale*).

Range Issues:

To ensure proper functionality, arguments must be within specified limits:

u_dc_bus must be within the range:

0 <= *u_dc_bus* <= 1 for 0% — 100% of maximum voltage

Amplitude must be within the fractional range:

0 <= *AmplitudeVoltScale* <= 1 for 0% — 100% of maximum voltage

AmplitudeAmplScale must be within the fractional range:

0 <= *AmplitudeAmplScale* <= 1 for 0% — 100% of sine amplitude

All input values must be in interval <0, max. UByte value>

Special Issues:

The *mcgenRippleCancel* function is intended to be used periodically; (e.g., called from within a timer interrupt or PWM update interrupt). This function usually precedes the 3-phase waveform generation function.

Design/Implementation:

The *mcgenRippleCancel* function is implemented as a library function call.

Performance:

See **Table 3-3**.

Table 3-3. *mcgenRippleCancel* Performance

Code size	35 B
Execution cycles	150 c

Code Examples:

See **Example 5 ". mcgen3PhWaveSine"** and **Example 6 ". mcgen3PhWaveSine3rdH"**

3.4.2 mcgen3PhWaveSine — 3-Phase Sine Wave

Call(s):

```
void mcgen3PhWaveSine (UByte          Amplitude,
                       SWord16        ActualPhase,
                       mc_s3PhaseSystem *sPhaseVoltage);
```

Parameters:

See **Table 3-4**.

Table 3-4. *mcgen3PhWaveSine* Parameters

Variable	Direction	Range	Explanation
PhaseA	out	0..0xFFFF	Percentage of the wave output value for Phase A voltage (0%..100%)
PhaseB	out	0..0xFFFF	Percentage of the wave output value for Phase B voltage (0%..100%)
PhaseC	out	0..0xFFFF	Percentage of the wave output value for Phase C voltage (0%..100%)
Amplitude	in	0..0xFF	Desired phase voltage amplitude
ActualPhase	in	0..0xFFFF	ActualPhase

Description:

The *mcgen3PhWaveSine* function from given *Amplitude* and *ActualPhase* calculates an immediate value of the three phase sinusoidal system:

- Phase A — *sPhaseVoltage.PhaseA*
- PhaseB — *sPhaseVoltage.PhaseB*
- Phase C — *sPhaseVoltage.PhaseC*

The individual waves are shifted 120° from each other. The shape of the generated waveforms depends on the data stored in the sine table. In motor control applications, data usually describes the pure sinewave or sinewave with the addition of a third harmonic component.

Figure 3-2 shows the duty cycles generated by the *mcgen3PhWaveSine* function when the *Amplitude* is 100%.

Figure 3-3 shows the duty cycles generated by the *mcgen3PhWaveSine* function when *Amplitude* is 50%.

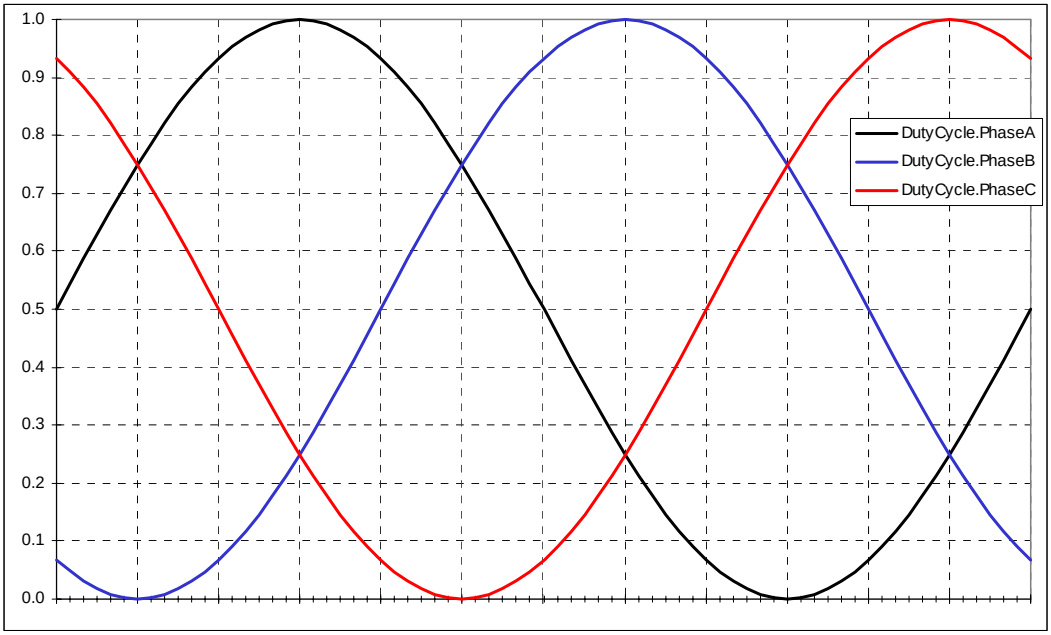


Figure 3-2. 3-Phase Sine Waves with 3rd Harmonic Injection *Amplitude* = 100%

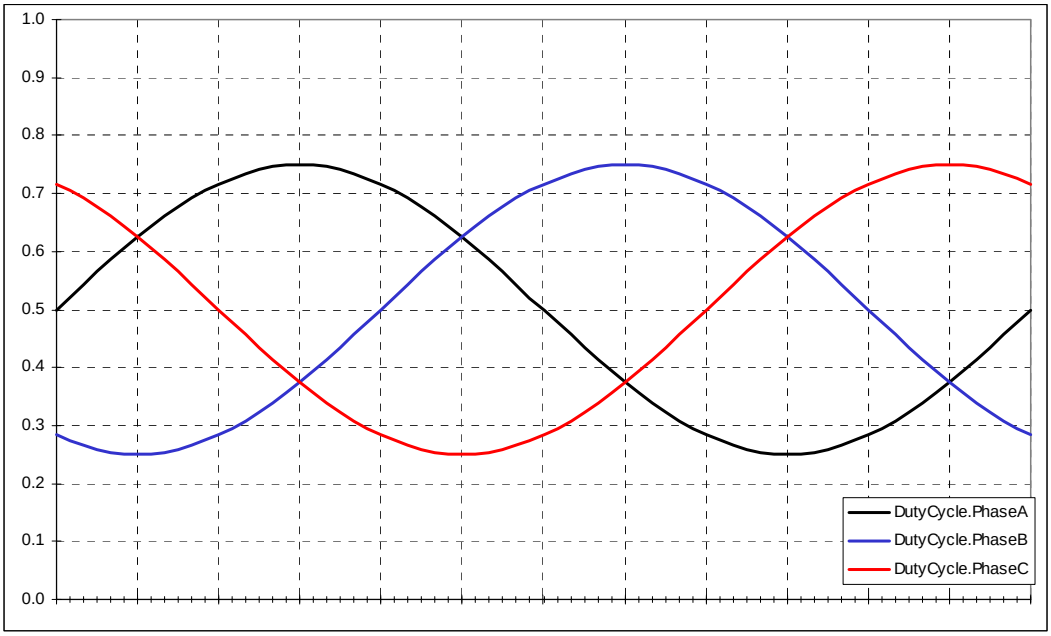


Figure 3-3. 3-Phase Sine Waves with 3rd Harmonic Injection *Amplitude* = 50%

Returns:

Modifies the output structure pointed to by *sPhaseVoltage.*

Range Issues:

To ensure proper wave generation, arguments must be within the following limits.

Amplitude must be within the fractional range:

0x00 <= *Amplitude* <= 0xFF for 0%..100% of generated sine amplitude

The increment (PhaseIncrement) of the ActualPhase added to each function call should be at most 1/16 of whole sine period in order to generate smooth sinewaves.

0x800 <= *ActualPhase* <= 0x7FF for — 100%..100% of sine period (0x8000..0x7FFF)

Special Issues:

The *mcgen3PhWaveSine* function is intended to be used periodically, (e.g., called from within a timer interrupt or PWM update interrupt). The input parameter *PhaseIncrement* must be calculated properly with respect to the *mcgen3PhWaveSine* function access period in order to generate the correct sine wave frequency.

Example:

Access frequency of function *mcgen3PhWaveSine* based on interrupt frequency = 4 kHz

Desired generated sine wave frequency = 100 Hz

PhaseIncrement must be set to 1638 (*65535*100/4000 = 1638)*

Design/Implementation:

The *mcgen3PhWaveSine* is implemented as a library function call.

Performance:

See **Table 3-5**.

Table 3-5. *mcgen3PhWaveSine* Performance

Code size (function)	*mcgen3PhWaveSine* + *Sine* 110 B + 34 B
Code size (table)	256 B
Execution cycles	288 c

Example 5. mcgen3PhWaveSine

```
/*********************************************************************/
/*                      I N C L U D E                                */
/*********************************************************************/
#include "types.h"     /* Generic SDK types */
#include "mcgen.h"     /* Waveform generation library */

/*********************************************************************/
/*                      GLOBAL STATIC VARIABLES                      */
/*********************************************************************/
static mc_s3PhaseSystem sPhaseVoltage; /* generated phase voltage
                                          amplitudes passed to PWM driver */
static SWord16         PhaseIncr;
static SWord16         ActualPhase;
static UByte           u_dc_bus     /* DC - bus voltage */
static UByte           u_ramp;      /* Output voltage from ripple
                                       cancelation function */
static UByte           Amplitude;   /* Amplitude of sinewaves
                                       (in % of max. phase voltage ampl.) */
static UByte           Voltage;     /* desired phase voltage
                                       in the level of dc-bus volt. */

/*********************************************************************/
/*                      FUNCTION PROTOTYPES                          */
/*********************************************************************/

void pwm_Reload_ISR (void);   /* PWM reload interrupt callback */

void main (void)
{
/* initialization of PWM driver */
.
.
EnableInterrupts();           /* Enable ISR */


PhaseIncr = 819;              /* defines sine frequency */
Amplitude = 128;              /* 50% sine amplitude */

while(1);                     /* endless loop */
}

void pwm_Reload_ISR (void)    /* PWM Interrupt subroutine */
{
ActualPhase += PhaseIncrement; /* new value of ActualPhase */

u_ramp = mcgenRippleCancel(Amplitude, u_dc_bus);

/* calculates the phase voltages for individual phases */
mcgen3PhWaveSine (u_ramp, ActualPhase, &sPhaseVoltage);

/* call PWM Driver function */
.
.
}
```

3.4.3 mcgen3PhWaveSine3rdH — 3-Phase Sine Wave with Third Harmonic

Call(s):

```
void mcgen3PhWaveSine3rdH (UByte      Amplitude,
                           SWord16    ActualPhase,
                           mc_s3PhaseSystem*sPhaseVoltage);
```

Table 3-6. *mcgen3PhWaveSine3rdH* Parameters

Variable	Direction	Range	Explanation
PhaseA	out	0..0xFFFF	Percentage of the wave output value for Phase A voltage (0%..100%)
PhaseB	out	0..0xFFFF	Percentage of the wave output value for Phase Bvoltage (0%..100%)
PhaseC	out	0..0xFFFF	Percentage of the wave output value for Phase C voltage (0%..100%)
Amplitude	in	0..0xFF	Desired phase voltage amplitude
ActualPhase	in	0..0xFFFF	ActualPhase

Description:

The *mcgen3PhWaveSine3rdH* function from given *Amplitude* and *ActualPhase* calculates an immediate value of the three phase sinusoidal system:

- Phase A — *sPhaseVoltage.PhaseA*
- PhaseB — *sPhaseVoltage.PhaseB*
- Phase C — *sPhaseVoltage.PhaseC*

The individual waves are shifted 120° from each other. The shape of the generated waveforms depends on the data stored in the sine table. In motor control applications, data usually describes the pure sinewave or sinewave with addition of third harmonic component. shows the duty cycles generated by the *mcgen3PhWaveSine3rdH* function when *Amplitude* is 100%.

Figure 3-4 shows the duty cycles generated by the *mcgen3PhWaveSine3rdH* function when the *Amplitude* is 100%.

Figure 3-5 shows the duty cycles generated by the *mcgen3PhWaveSine3rdH* function when the *Amplitude* is 50%.

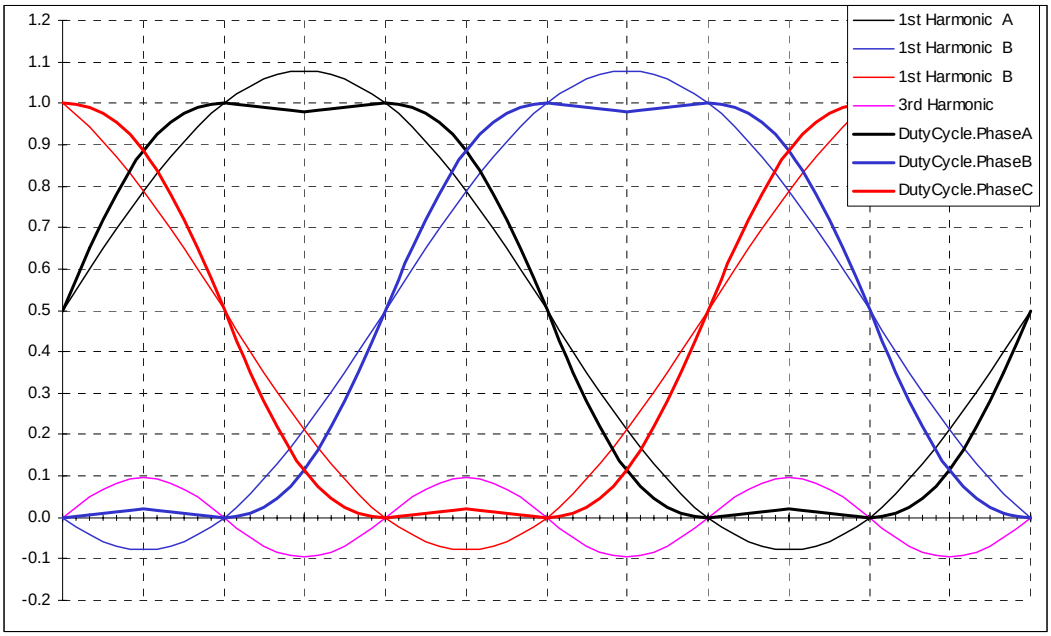


Figure 3-4. 3-Phase Sine Waves with 3rd Harmonic Injection
***Amplitude* = 100%**

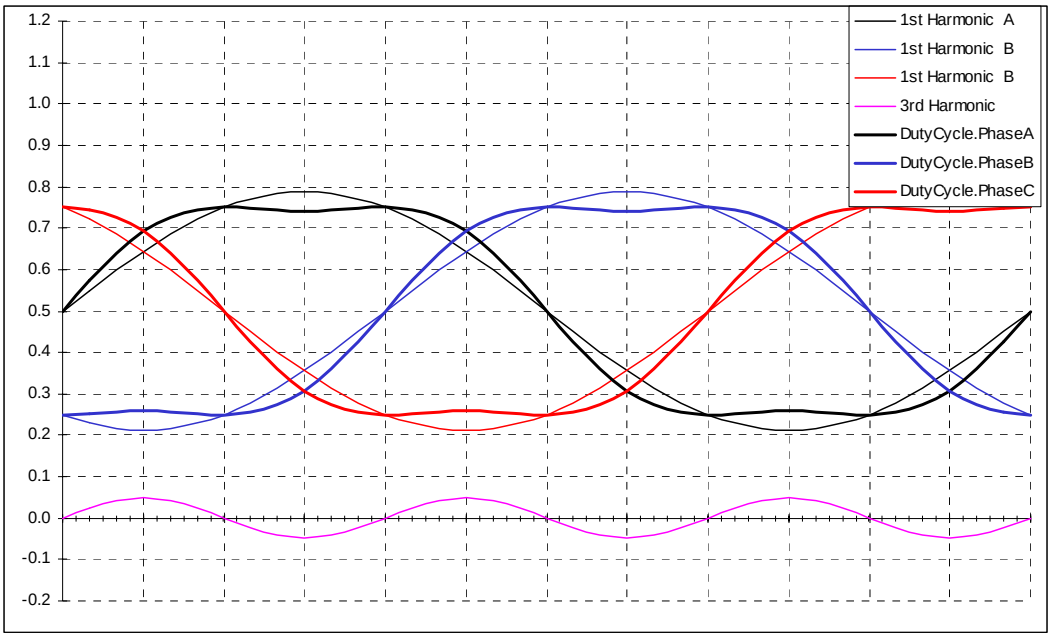


Figure 3-5. 3-phase Sine Waves with 3rd Harmonic Injection
***Amplitude* = 50%**

Returns:

Modifies the output structure pointed by *sPhaseVoltage.*

Range Issues:

To ensure proper wave generation, arguments must be within the following limits:

Amplitude must be within the fractional range:

0x00 <= *Amplitude* <= 0xFF for 0%..100% of generated sine amplitude

The increment (PhaseIncrement) of the ActualPhase added each function call should be at most 1/16 of whole sine period in order to generate smooth sinewaves.

0x800 <= *ActualPhase* <= 0x7FF for -100%..100% of sine period (0x8000..0x7FFF)

Special Issues:

The *mcgen3PhWaveSine3rdH* function is intended to be used periodically (e.g., called from within a timer interrupt or PWM update interrupt). The input parameter *PhaseIncrement* must be calculated properly with respect to the *mcgen3PhWaveSine3rdH* function access period in order to generate the correct sine wave frequency.

Example:

Access frequency of function *mcgen3PhWaveSine3rdH* based on interrupt frequency = 4 kHz

Desired generated sine wave frequency = 100 Hz

PhaseIncrement must be set to 1638 (*65535*100/4000 = 1638)*

Design/Implementation:

The *mcgen3PhWaveSinerdH* is implemented as a library function call.

Performance:

Table 3-7. *mcgen3PhWaveSine3rdH* Performance

Code size (function)	*mcgen3PhWaveSine3rdH + Sine3rdH*. 122 B + 34 B
Code size (table)	256 B
Execution cycles	329 c

Example 6. mcgen3PhWaveSine3rdH

```
/*****************************************************************/
/*                    I N C L U D E                              */
/*****************************************************************/
#include "types.h"     /* Generic SDK types */
#include "mcgen.h"     /* Waveform generation library */

/*****************************************************************/
/*                    GLOBAL STATIC VARIABLES                    */
/*****************************************************************/
static mc_s3PhaseSystem sPhaseVoltage; /* generated phase voltage
                                          amplitudes passed to PWM driver */
static SWord16         PhaseIncr;
static SWord16         ActualPhase;
static UByte           u_dc_bus     /* DC - bus voltage */
static UByte           u_ramp;      /* Output voltage from ripple
                                       cancelation function */
static UByte           Amplitude;   /* Amplitude of sinewaves
                                       (in % of max. phase voltage ampl.) */
static UByte           Voltage;     /* desired phase voltage
                                       in the level of dc-bus volt. */

/*****************************************************************/
/*                    FUNCTION PROTOTYPES                        */
/*****************************************************************/

void pwm_Reload_ISR (void);   /* PWM reload interrupt callback */

void main (void)
{
/* initialization of PWM driver */
.
.
EnableInterrupts();           /* Enable ISR */


PhaseIncr = 819;              /* defines sine frequency */
Amplitude = 128;              /* 50% sine amplitude */

while(1);                     /* endless loop */
}

void pwm_Reload_ISR (void)    /* PWM Interrupt subroutine */
{
ActualPhase += PhaseIncrement; /* new value of ActualPhase */

u_ramp = mcgenRippleCancel(Amplitude, u_dc_bus);

/* calculates the phase voltages for individual phases */
mcgen3PhWaveSine3rdH (u_ramp, ActualPhase, &sPhaseVoltage);

/* call PWM Driver function */
.
.
```

Section 4. Volts-per-Hertz (V/Hz) Table

4.1 Contents

4.2 Introduction

This section describes the the Application Programming Interface (API) for the volts-per-hertz (V/Hz) table.

4.3 API Definitions

The header files *types.h* and *vhz.h* include all required prototypes and structure/type definitions. This information is included here for the programmer's reference.

Public Interface Function(s):

```
/*****************************************************************************
*
*   MAKRO: VHZ_CREATE_TABLE ()
*
*   DESCRIPTION: This macro calculates the parameters of V/Hz table from given
*                parameters.
*
*   EXAMPLE:
*   #define V_BOOST     20          /* 0 .. 100 */
*   #define V_BASE      80          /* 0 .. 100 */
*   #define F_BOOST      5          /* 0 .. 100 */
*   #define F_BASE      40          /* 0 .. 100 */
*
*   const vhz_sTable vhzTable = VHZ_MAKE_TABLE(V_BOOST, V_BASE, F_BOOST, F_BASE);
*
*   RETURNS:  vhz_sTable type structure with V/Hz data elements.
*
*   ARGUMENTS:
*       UByte   v_boost     boost voltage in % to max. phase voltage amplitude
*       UByte   v_base      base voltage  in % to max. phase voltage amplitude
*       SByte   f_boost     boost frequency in % to max. generated output frequency
*       SByte   f_base      base frequency in  % to max. generated output frequency
*
*   RANGE ISSUES:   None
*
*       v_base - v_boost
*       ---------------  < 4   to make sure that slope will not overflow
*       f_base - f_boost
*
*   SPECIAL ISSUES:
*
*****************************************************************************/

const vhz_sTable vhzTable = VHZ_CREATE_TABLE(v_boost, v_base, f_boost, f_base);

/*****************************************************************************
*
*  MODULE: vhzGetVoltage()
*
*  DESCRIPTION: Function calculates the required phase voltage amplitude
*               based on V/Hz motor specific parameters and required frequency
*
*  RETURNS:
*       UByte                   required phase voltage amplitude
*
*  ARGUMENTS:
*       vhz_sTable *vhzTable     V/Hz table pointer
*       SWord16     frequency    required frequency
*
*  RANGE ISSUES:   None
*
*  SPECIAL ISSUES: None
*
*****************************************************************************/

UByte vhzGetVoltage(vhz_sTable *vhzTable, SWord16 frequency);
```

Public Data Structure(s):

Data structure vhz_sTable `is` defined in vhz.*h* header file. See **Table 4-1**.

```
typedef struct {
    UByte v_boost;
    UByte v_base;
    UByte f_boost;
    UByte f_base;
    UByte slope;
} vhz_sTable;
```

Table 4-1. *vhz_sTable* Structure Elements

Variable	Explanation
v_boost	Boost voltage (0..100%) of the max. phase voltage amplitude
v_base	Base voltage (0..100%) of the max. phase voltage amplitude
f_boost	Boost frequency (0..100%) of the max. output frequency
f_base	Base frequency (0..100%) of the max. output frequency
slope	Slope of V/Hz ramp between Boost and Base point

4.4 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

4.4.1 VHZ_CREATE_TABLE — Create the V/Hz Table

Call(s):

```
vhz_sTable vhzTable = VHZ_CREATE_TABLE(v_boost, v_base,
f_boost, f_base);
```

Parameters:

See **Table 4-2**.

Table 4-2. *VHZ_CREATE_TABLE* Parameters

Variable	Direction	Explanation
v_boost	in	Boost voltage (0..100%) of the maximum phase voltage amplitude
v_base	in	Base voltage (0..100%) of the maximum phase voltage amplitude
f_boost	in	Boost frequency (0..100%) of the maximum output frequency
f_base	in	Base frequency (0..100%) of the maximum output frequency

Description:

The *VHZ_CREATE_TABLE* macro calculates the parameters of a volt per hertz (V/Hz) table from parameters which are set in:

- Percentages related to maximal phase voltage amplitude
- Or Percentages maximal output frequency.

Returns:

Modifies the output structure of **vhz_sTable type**

Range Issues:

To ensure correct calculation of the V/Hz ramp parameters the following condition must be fulfilled. Otherwise, the slope parameter may overflow.

$$\frac{v\_base - v\_boost}{f\_base - f\_boost} < 4$$

Special Issues:

The macro **VHZ_CREATE_TABLE** must be called before the function **vhzGetVoltage** is used to ensure proper functionality.

Design/Implementation:

The **VHZ_CREATE_TABLE** is implemented as a macro.

4.4.2 vhzGetVoltage — Calculate the Phase Voltage Amplitude

Call (s):

```
UByte vhzGetVoltage(vhz_sTable *vhzTable, SWord16 frequency);
```

Parameters:

See **Table 4-3**.

Table 4-3. vhzGetVoltage Parameter

Variable	Direction	Explanation
vhzTable	in	structure of vhz_sTable type with volt-per-hertz parameters
frequency	in	electrical frequency applied to the motor <-32768, 32767> ~ <-max_freq , max_frex>

Description:

The algorithm **vhzGetVoltage** returns voltage for the required frequency according to a defined table. The volt-per-hertz control method is the most popular method of scalar control and controls the magnitude of variables such as frequency, voltage, or current. The command and feedback signals are DC quantities which are proportional to the respective variables.

This scheme is defined as volt-per-hertz control because the voltage applied command is calculated directly from the applied frequency in order to maintain the air-gap flux of the machine constant. In steady state operation, the machine air-gap flux is approximately proportional to the ratio Vs/fs, where Vs is the amplitude of motor phase voltage and fs is the synchronous electrical frequency applied to the motor. The characteristic is defined by base point and boost point. In **Figure 4-1**, the boost point the minimal boost voltage is required to maintain the motor excited at the startup. Between the boost point and the base point the motor operates at optimum excitation, called constant torque operation, because of the constant Vs/fs ratio. Above this point the motor operates under-excited, called constant power operation, because of the rated voltage limit.

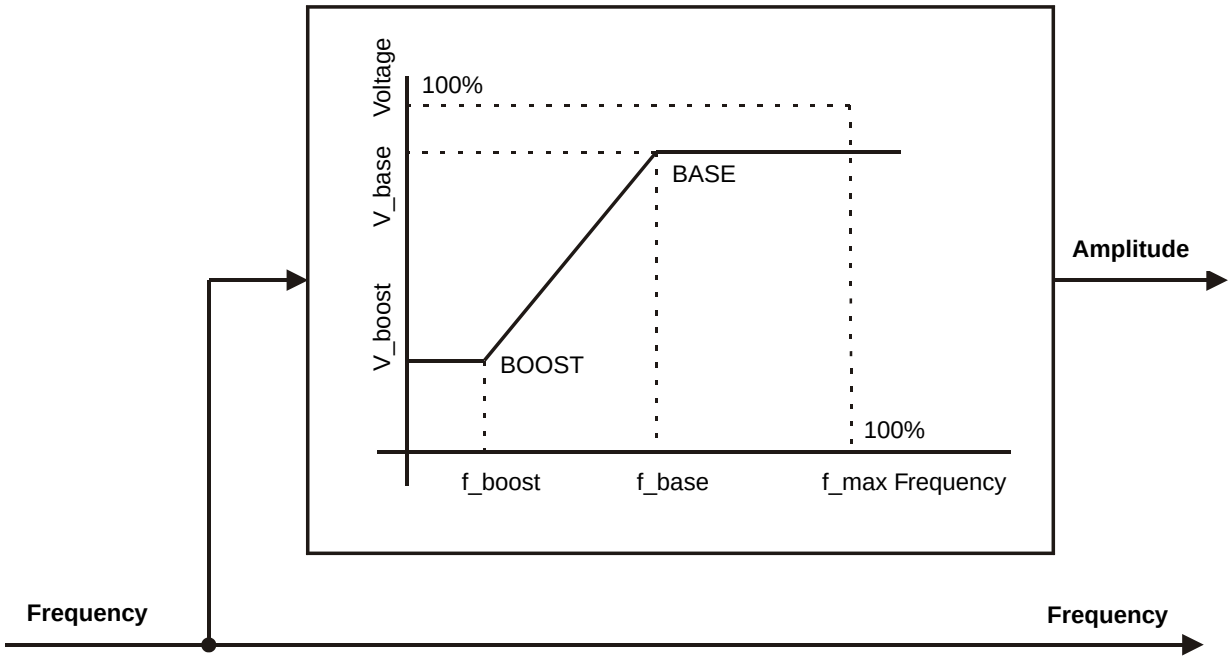


Figure 4-1. Volt-per-Hertz Characteristics

Returns: UByte

Range Issues: None

Special Issues:

The function **vhzGetVoltage** must be called before the function **vhzGetVoltage** is used to ensure proper functionality.

Design/Implementation:

vhzGetVoltage is implemented as a macro.

Performance:

Table 4-4. vhzGetVoltage Performance

Code size (function)	57 B
Code size (table)	5 B
Execution cycles	minimum/maximum/typical 96 / 147 / 147

Example 7. Volt-per-Hertz Table Usage

```
/*****************************************************************/
/*                      I N C L U D E                            */
/*****************************************************************/
#include "types.h"     /* Generic SDK types */
#include "vhz.h"       /* Volt per Hertz Ramp */

/* Parameters of Volt per Hertz specific to chozen induction motor */
#define V_BOOST                    17       /* [%] to nominal voltage */
#define V_BASE                     84       /* [%] to nominal voltage */
#define F_BOOST                     7       /* [%] to maximal speed */
#define F_BASE                     46       /* [%] to maximal speed */


/* PUT FOLLOWING CONSTANTS INTO FLASH */
#pragma CONST_SEG CONST_ROM

const vhz_sTable vhzTable = VHZ_CREATE_TABLE(V_BOOST, V_BASE, F_BOOST, F_BASE);

#pragma CONST_SEG DEFAULT

static mc_s3PhaseSystem sPhaseVoltage; /* generated phase voltage
amplitudes passed to
PWM driver */

static UByte           Amplitude;   /* Amplitude of sinewaves
(in % of max. phase voltage
ampl.) */
static SWord16         Phase        /* Actual phase of the stator phase voltage vector */
static UByte           u_dc_bus     /* DC bus voltage */
static UByte           u_ramp;      /* desired phase voltage
in the level of dc-bus volt. */

void main (void)
{
/* initialization of PWM driver */
EnableInterrupts();           /* Enable ISR */
PhaseIncr = 819;              /* defines sine frequency */
Amplitude = 0x3fff;           /* 50% sine amplitude */
while(1);                     /* endless loop */
}


void pwm_Reload_ISR (void)    /* PWM Interrupt subroutine */
{

/* calculates the */
u_ramp = vhzGetVoltage(&vhzTable, frequency);

/* removes the dc-bus voltage ripples from the phase voltage amplitude */
Amplitude = mcgenRippleCancel(u_ramp, u_dc_bus);


ActualPhase += PhaseIncrement; /* new value of ActualPhase */
/* calculates the phase voltages for individual phases */
mcgen3PhWaveSine (Amplitude, Phase, &sPhaseVoltage);

/* call PWM Driver function */
}
```

Volts-per-Hertz (V/Hz) Table

Section 5. Dead-Time Distortion Correction Algorithm

5.1 Contents

5.2 Introduction

This section describes the Application Programming Interface (API) for the dead-time correction algorithm and the detailed algorithm description.

5.3 Dead-Time Distortion Correction

Six-transistor inverter is the most used topology for AC motor drives. The dead time must be inserted between the turning off of one transistor in the inverter half bridge and turning on of the complementary transistor. The dead time causes distortion to the generated voltage, and thus a non-sinusoidal phase current.

In order to achieve a sinusoidal phase current, and thus limit the harmonic losses, noise, and torque ripple, the dead-time distortion correction needs to be implemented. The on-chip Pulse-Width-Modulation (PWM) module, of the MC68HC908MRxx Family of Motorola

microcontrollers, contains the patented hardware block that simplifies the task.

The dead-time correction is based on the evaluation of the phase current polarity of the respective phase, and proper counter modulation of the dead-time distortion. The basic situation is shown in **Figure 5-1**. The desired load voltage is affected by the dead time. During dead time, load inductance defines the voltage needed to keep inductive current flowing through diodes. So full positive or full negative voltage is applied to the phase, according to the phase current polarity. For positive current (i+), the actual voltage pulses are shortened by dead time, for negative phase current the voltage pulses are lengthened by dead time.

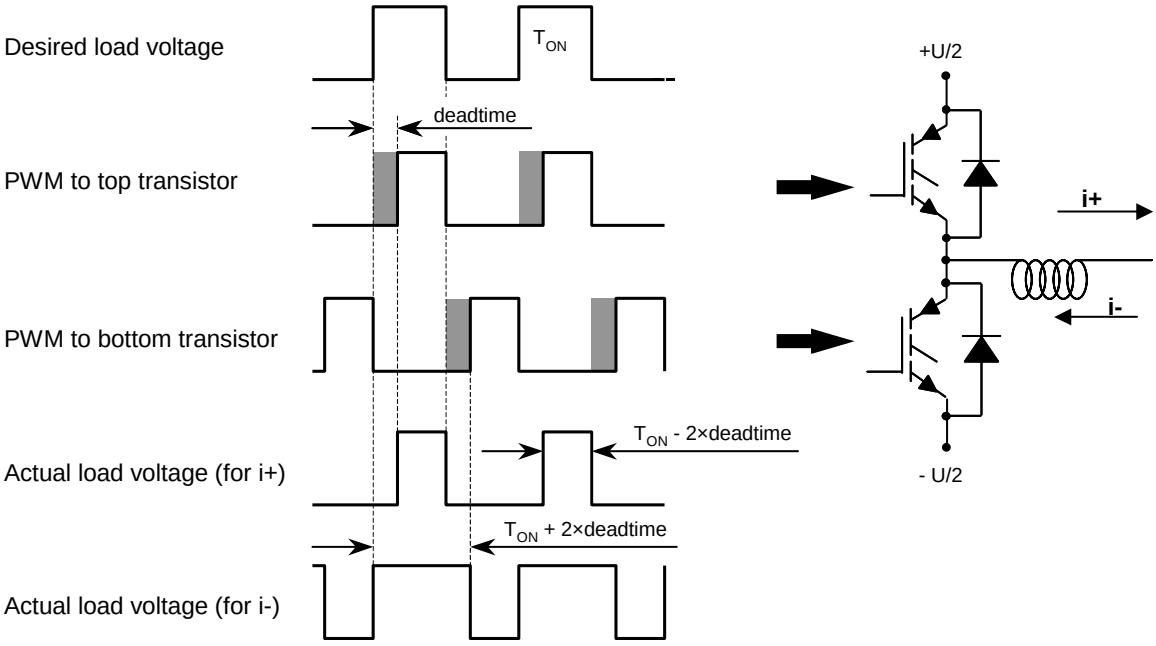


Figure 5-1. Dead-Time Distortion

To achieve distortion correction, one of two different correction factors must be added to the desired PWM value, depending on whether the top or bottom transistor is controlling the output voltage during the dead time.

When the voltage pulse is shortened due to dead time, the control PWM signal is extended by dead time, so the actual voltage pulse matches the

desired voltage. Vice versa, when the voltage pulse is lengthened due to dead time, the control PWM signal is shortened by dead time, so again the actual voltage pulse matches the desired voltage. Therefore, the actual signal equals the desired one, and the generated phase current is sinusoidal.

The dead-time distortion correction utilizes phase current sensing. The on-chip PWM module of MC68HC908MRxx microcontrollers contains the block that enables it to evaluate the polarity and the size of the phase current without the need of an expensive current sensor. It is based on the sampling and evaluation of the phase voltage level during the dead time. The zero voltage during dead time reflects a positive phase current, the full DC-Bus voltage during dead time reflects a negative phase current. So comparing the phase voltage with the half DC-Bus voltage, enables to evaluation of the current polarity. The topology is illustrated in **Figure 5-2**. The output of the comparator is connected to the current polarity sensing input of the microcontroller MC68HC908MR32. The microcontroller contains the hardware that samples the current sensing inputs during dead time. It enables evaluation of the current polarity and also the region of low currents.

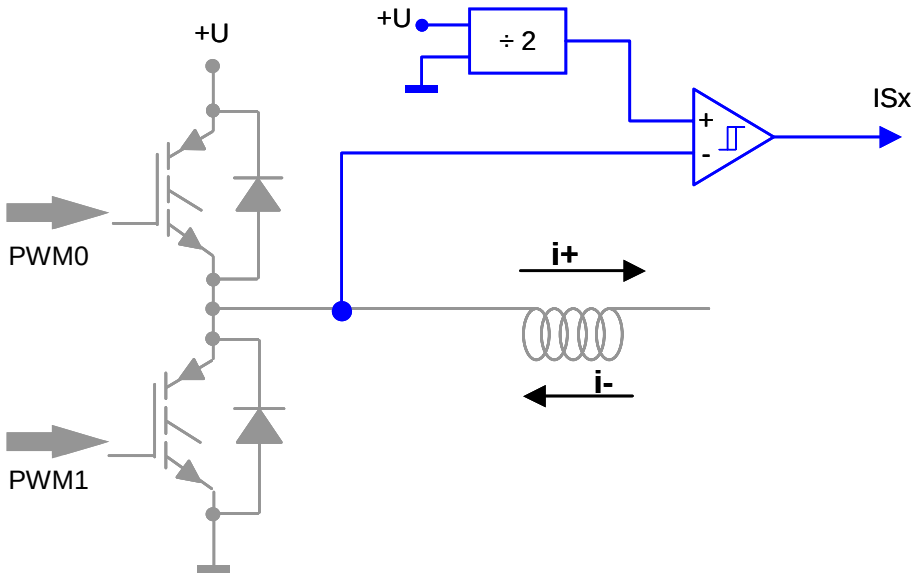


Figure 5-2. Topology of Current Polarity Sensing

During PWM reload ISR, the desired PWM values for all three phases are calculated as:

- PWM1 for phase 1
- PWM2 for phase 2
- PWM3 for phase 3

The values loaded into the individual PVAL registers of the separate phases are shown in **Table 5-1**. Since AC motor control utilizes center-aligned PWM modulation, only half of the dead time needs to be added to or subtracted from the desired PWM duty cycle to achieve the distortion correction. Without dead-time correction, the even PVAL registers are loaded with the required PWM value, but the odd PVAL registers are not used. When dead-time correction is used, the even PVAL registers are loaded with the desired PWM *plus* half of the dead time (PWMx+DT/2), while the odd PVAL registers are loaded with the desired PWM *minus* half of the dead time (PWMx–DT/2).

When calculating the values to be loaded into the PVAL registers, the MRxx's dead time register can be used.

The dead-time register (DEADTM) holds an 8-bit value which specifies the number of CPU clock cycles to be used for the dead-time, when complementary PWM mode is selected. Dead-time is not affected by changes to the prescaler value. On the other hand, the PVAL values are affected by the prescaler of the PWM counter.

Table 5-1. PWM Values Loaded into Registers PVAL1–PVAL6

Phase	PVAL Register	Required Values in PVAL without Dead-Time Correction	Required Values in PVAL with Dead-Time Correction	Actual Values Loaded into PVAL Registers
Phase 1	PVAL1	PWM1	PWM1 + DT/2	PWM1 + DEADTM/2/PWM_PRESC
	PVAL2	—	PWM1 – DT/2	PWM1 - DEADTM/2/PWM_PRESC
Phase 2	PVAL3	PWM2	PWM2 + DT/2	PWM2 + DEADTM/2/PWM_PRESC
	PVAL4	—	PWM2 – DT/2	PWM2 - DEADTM/2/PWM_PRESC
Phase 3	PVAL5	PWM3	PWM3 + DT/2	PWM3 + DEADTM/2/PWM_PRESC
	PVAL6	—	PWM3 – DT/2	PWM3 - DEADTM/2/PWM_PRESC

Therefore, the value stored into the dead time registers needs to be scaled by the PWM prescaler (PWM_PRESC in **Table 5-1**). The PWM Control Register 2 (PCTL2) contains the PWM generator prescaler. The buffered read/write bits, PRSC0 and PRSC1, select the PWM prescaler according to **Table 5-2**.

Table 5-2. PWM Prescaler

Prescaler Bits PRSC0 and PRSC1	PWM Frequency	Prescaler PWM_PRESC
00	f_{OP}	1
01	$f_{OP}/2$	2
10	$f_{OP}/4$	4
11	$f_{OP}/8$	8

The on-chip PWM module of MC68HC908MRxx microcontrollers enables it to perform two types of dead-time distortion correction:

- **Partial** correction
- **Full** correction

Partial dead-time distortion correction is based only on polarity detection of phase current. The hardware, sensing the current polarity according to **Figure 5-2**, needs to be implemented. The software is responsible for calculating both compensated PWM values and placing them in an odd/even PWM register pair according to **Table 5-1**. The distortion correction is fully implemented by the on-chip PWM module according to the following scheme:

- If the **current** sensed at the motor for that PWM pair **is positive** (voltage on current pin ISx is low), **the odd PWM value is used** for the PWM pair.
- Likewise, if the **current** sensed at the motor for that PWM pair **is negative** (voltage on current pin ISx is high), **the even PWM value is used**.

For partial correction, the on-chip dead-time correction block is set in the automated mode — current sense correction bits ISENS1:ISENS0 of PWM Control Register 0 (PCTL1) are set to 10.

The disadvantage of the partial correction is that some dead-time distortion still exist — the current is flattened out at the zero crossings.

Full dead-time distortion correction (implemented in **dtCorrectFull algorithm**) improves the partial dead-time correction by sensing not only the polarity, but also the magnitude of the actual phase current.

In the full dead-time correction method, the threshold where the correction values should be toggled is not in the zero level, but slightly advanced. The threshold is illustrated in **Figure 5-3**. Toggling of the correction offset needs to occur before the current has a chance to flatten out at a current zero-crossing. So, the current sense scheme must sense that the current waveform is approaching the zero-crossing.

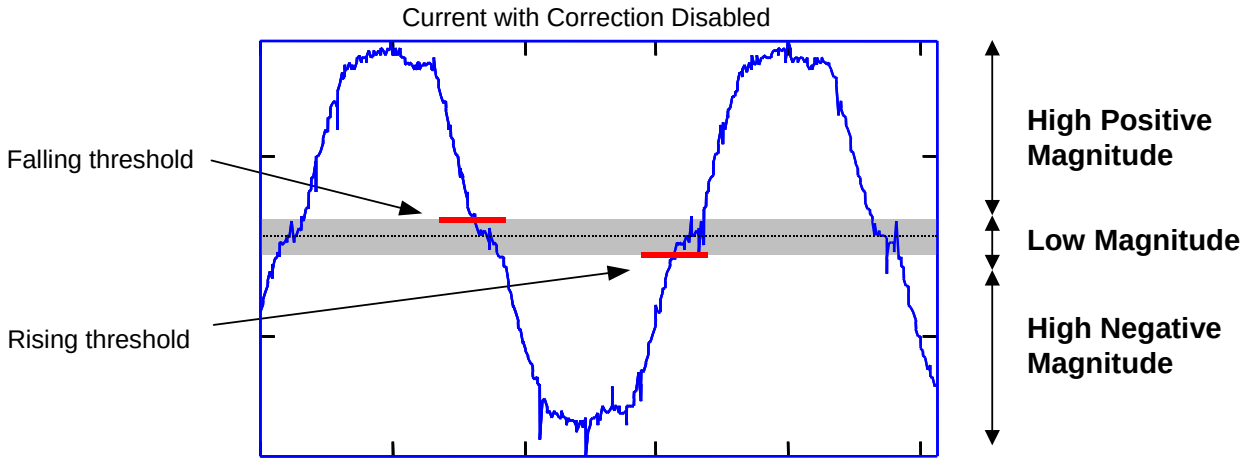


Figure 5-3. Proposed Current Threshold for Correction Toggling

To achieve the full distortion correction, again one of two different correction factors must be added to the desired PWM value, depending on whether the top or bottom transistor is controlling the output voltage during the dead time. The software is responsible for calculating both compensated PWM values and placing them in an odd/even PWM register pair. Then the software needs to determine which PWM value is to be used, according to the following scheme:

- If the **current** sensed at the motor for that PWM pair **is positive and of high magnitude,** or **negative and of small magnitude in a trend approaching zero crossing**, **the odd PWM value is used** for the PWM pair.
- Likewise, if the **current** sensed at the motor for that PWM pair **is negative,** or **positive and of small magnitude in a trend approaching zero crossing**, **the even PWM value is used**.

The MR32 contains a hardware circuitry that enables it to sense the current polarity together with the magnitude. The current polarity and magnitude is sensed using the DT-DT6 of FTACK register in the MC68HC908MR32 microcontroller. For Phase 1, the bits DT1 and DT2 are used as shown in **Table 5-3**.

Table 5-3. Sensing of the Current Polarity and Magnitude for Phase 1

DT1	DT2	Current Condition of Phase 1
0	0	High magnitude I+
1	1	High magnitude I–
0	1	Low magnitude, either polarity

For phase 2, bits DT3 and DT4 are used. For phase 3, bits DT5 and DT6 are used.

As was stated the determination of the correct PVAL used for the PWM generation is done purely by software. The on-chip dead-time correction block is set in the manual mode — current sense correction bits ISENS1:ISENS0 of PWM Control Register 0 (PCTL1) are set to 00 or 01.

5.4 API Definitions

The header files *types.h* and *dtCorrect.h* include all required prototypes and structure/type definitions. This information is included here for the programmer's reference.

Public Interface Function(s):

```
/*****************************************************************************
*
*  MODULE: dtCorrectInit ()
*
*  DESCRIPTION:
*      Function initializes the structure of the dead time correction algorithm
*
*  RETURNS:
*      dtStateFlagsAB = 0     initialize dead-time correction flags phases A & B
*      dtStateFlagsC = 0      initialize dead-time correction flags phase C
*      pointA = 0       initialize internal capture of the pointer for ph. A
*      pointB = 0       initialize internal capture of the pointer for ph. B
*      pointC = 0       initialize internal capture of the pointer for ph. C
*      ipolBits = 0     clear the internal IPOL bits
*
*  ARGUMENTS:
*      dtCorrect_s *pDtCorrect pointer to dead time correction structure
*
*  RANGE ISSUES:   None
*
*  SPECIAL ISSUES:
*      The function dtCorrectInit must be called before the function
*      dtCorrectFull starts to be called to ensure proper functionality
*
*****************************************************************************/

void dtCorrectInit (dtCorrect_s *pDtCorrect);

/*****************************************************************************
*
* MODULE: dtCorrectFull ()
*
* DESCRIPTION:
*    Function calculates the IPOL bits, defining the PVAL registers to be
*    used for PWM generation for optimized dead time correction.
*    The bits are determined according to the phase current polarity detection
*    bits DT1-6, actual sine wave pointer and the actual state of the algorithm
*    state machine.
*
* RETURNS:
*    dtStateFlagsAB    updates internal flag register for phases A and B
*                      according to the state of the algorithm state machine
*    dtStateFlagsC     updates internal flag register for phase C
*                      according to the state of the algorithm state machine
*    pointA            up-dates internal capture of the pointer for ph. A
*    pointB            up-dates internal capture of the pointer for ph. B
*    pointC            up-dates internal capture of the pointer for ph. C
*    ipolBits          updates the IPOL bits according to the actual current
*                      polarity and the state of the algorithm state machine
```

```
*
* ARGUMENTS:
*    dtCorrect_s *pDtCorrect pointer to dead time correction structure
*
* RANGE ISSUES:
*    The dead time correction adds the correction factor to the originally
*    calculated sine waves. It is necessary to ensure that the calculated PWM
*    duty cycles do not exceed the PWM modulus.
*
* SPECIAL ISSUES:
*      The function dtCorrectInit must be called before the function
*      dtCorrectFull starts to be called to ensure proper functionality
*
*******************************************************************************/

void dtCorrectFull (dtCorrect_s *pDtCorrect);
```

Public Data Structure(s):

Data structure `dtCorrect_s` is defined in *dtCorrect.h* header file.

See **Table 5-4**

```
typedef struct {
    UByte dtBits;
    UByte ipolBits;
    type_uBits dtStateFlagsAB;
    type_uBits dtStateFlagsC;
    SByte pointA;
    SByte pointB;
    SByte pointC;
    SByte pointerA;
} dtCorrect_s;
```

Table 5-4. dtCorrect_s Structure Elements

Variable	Explanation
dtBits	INPUT: actual status of the dead time bits DT1-6 format | x | x |DT6|DT5| |DT4|DT3|DT2|DT1| fits to FTACK of ‘MR32
ipolBits	OUTPUT: ipolBits - new top/bottom correction bits IPOL1-3 format | x | x | x |IPOL1| |IPOL2|IPOL3| x | x | fits to PCTL2 of ‘MR32
dtStateFlagsAB	Internal dead-time correction flags for phases AB
dtStateFlagsC	Internal dead-time correction flags for phase C
pointA	Internal capture of the pointer for phase A
pointB	Internal capture of the pointer for phase B
pointC	Internal capture of the pointer for phase C
pointerA	INPUT: actual pointer of the generated wave phase A

5.5 API Specification

This section specifies the exact usage of each API function.

Function arguments for each routine are described as in, out, or inout.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables, in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

5.5.1 dtCorrectInit () - Initialize Dead-Time Correction Algorithm

Call(s):

```
dtCorrectInit (dtCorrect_s *pDtCorrect);
```

Parameters:

See **Table 5-5**:

Table 5-5. dtCorrectInit Parameters

Variable	Direction	Explanation
dtCorrect_s	in, out	Structure of the dead-time correction algorithm

Description:

The function **dtCorrectInit** initializes the individual members of the dead-time correction structure to zero. The function **dtCorrectInit** must be called once before the dead-time correction algorithm is enabled.

Returns:

dtCorrect_s

Range Issues:

None

Special Issues:

The function **dtCorrectInit** must be called before starting any calls to the function **dtCorrectFull,** to ensure proper functionality.

Design/Implementation:

The **dtCorrectInit** is implemented as a function.

Performance:

Table 5-6. dtCorrectInit Performance

Code Size	57 B		
Execution Cycles	Minimum 96	Maximum 147	Typical 147

5.5.2 dtCorrectFull () - Perform Dead-Time Correction Algorithm

Call (s):

```
dtCorrectFull (dtCorrect_s *pDtCorrect);
```

Parameters:

See **Table 5-7**.

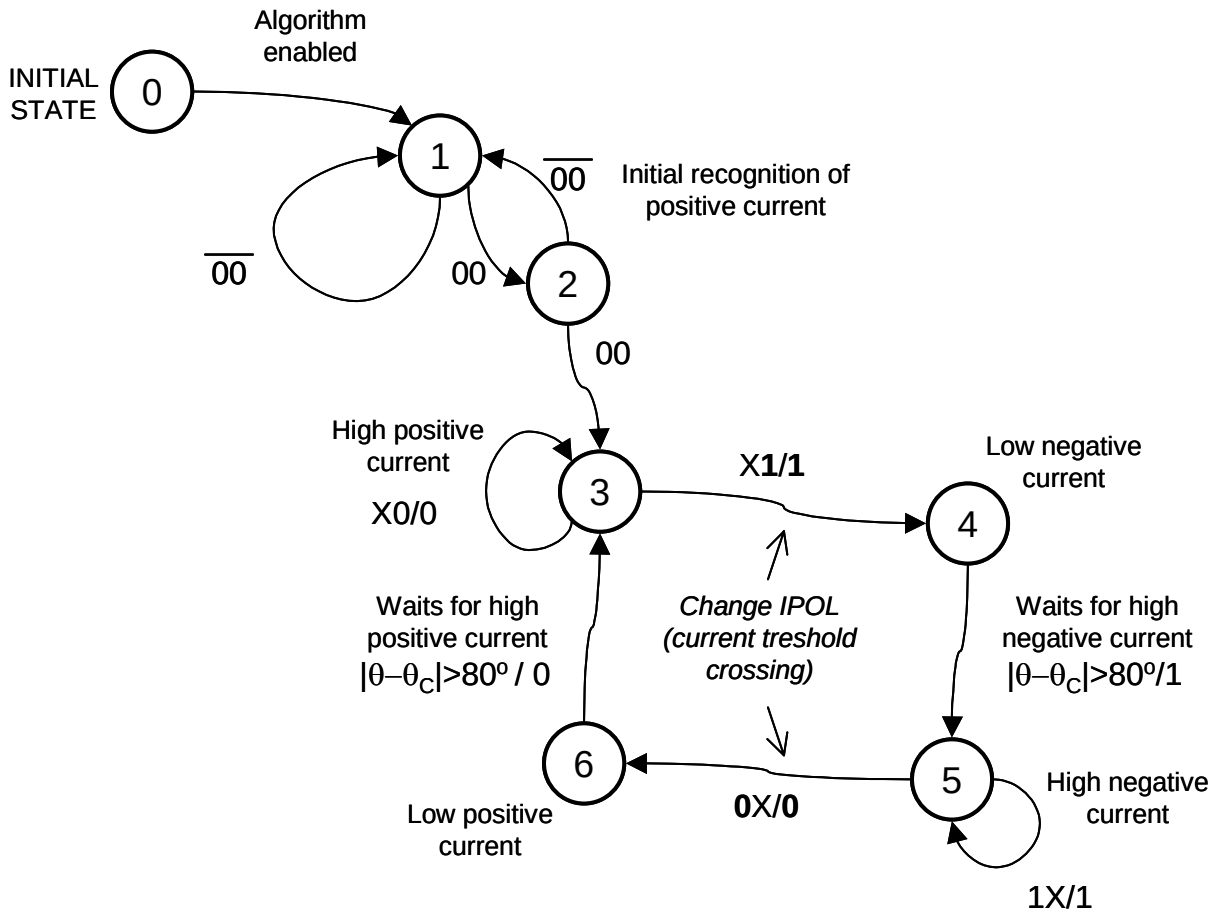
Table 5-7. dtCorrectFull Parameter

Variable	Direction	Explanation
dtCorrect_s	in, out	Structure of the dead-time correction algorithm

Description:

The algorithm **dtCorrectFull** calculates the IPOL bits, defining the PVAL registers to be used for MC68HC908MR32 PWM generation for full dead-time correction. The IPOL bits are determined according to the phase current polarity detection bits DT1–DT6, actual sine wave pointer, and the actual state of the algorithm state machine.

The algorithm state machine samples the actual state of the phase current, and selects the appropriates PVAL registers to be used for PWM generation. The state machine, implemented in the **dtCorrectFull** algorithm is illustrated in **Figure 5-4**.



STATE TRANSITION KEY: DT1 DT2 / IPOL
IPOL = 0: ODD NUMBERED PWM REGISTER CONTROLS OUTPUT
IPOL = 1: EVEN NUMBERED PWM REGISTER CONTROLS OUTPUT

Figure 5-4. Dead-Time Correction State Machine

When the algorithm is enabl

ed, the state machine is entered from initial state 0. It is waiting till the high magnitude of positive current is detected (State 1, confirmed by State 2), then the algorithm enters the state machine (State 3). The state machine is performed in circle 3-4-5-6-3. As soon as the low magnitude of negative current is detected, the IPOL is changed to 1, requesting the even numbered PWM registers to be used for PWM generation, the actual value of the wave pointer is recorded (θ_C), and State 4 is entered. State 4 is preserved for 85 electrical degrees, until a high negative current can be expected. Then State 5 is entered. As soon as the low magnitude of positive current is detected, the IPOL is changed to 0, requesting the odd numbered PWM registers to be used for PWM generation, the actual value of the wave pointer is recorded (θ_C), and State 6 is entered. State 6 is preserved for 85 electrical degrees, until a high positive current can be expected. Then State 3 is entered and the state machine loop is repeated. In this way it is ensured that the required IPOL changes when a small amplitude of respective current is detected by the hardware.

NOTE: *Please note, that the wave pointer is recorded into the algorithm variable PointA, PointB, or PointC, in the moment when the respective phase current crosses the low current threshold.*

Such a state machine is independently implemented for each phase (A, B, C). The algorithm contains two flag variables, determining the actual state of the state machine for individual phases. Flag variable dtStateFlagsAB determines state of state machine for phases A & B, StateFlagsC determines the state of the state machine for phase C.

The meaning of the individual bits of dtStateFlagsAB are listed in **Table 5-8**. The meaning of the individual bits of dtStateFlagsC are listed in **Table 5-9**.

Table 5-8. Meaning of State Machine Flag Registers dtStateFlagsAB

Phase	Bits	State					
		1	2	3	4	5	6
Phase A	Bit 0 – lock	0	0	1	1	1	1
	Bit 1	0	1				
	Bit 2			0	1	0	1
	Bit 3			0	0	1	1
Phase B	Bit 4 – lock	0	0	1	1	1	1
	Bit 5	0	1				
	Bit 6			0	1	0	1
	Bit 7			0	0	1	1

Table 5-9. Meaning of State Machine Flag Registers dtStateFlagsC

Phase	Bits	State					
		1	2	3	4	5	6
Phase C	Bit 0 — lock	0	0	1	1	1	1
	Bit 1	0	1				
	Bit 2			0	1	0	1
	Bit 3			0	0	1	1
Reserved	Bit 4	x	x	x	x	x	x
	Bit 5	x	x	x	x	x	x
	Bit 6	x	x	x	x	x	x
	Bit 7	x	x	x	x	x	x

NOTE: *A detailed explanation of the dead-time distortion correction can be found in the comprehensive application note of Motorola* ***AN1728 "Making Low-Distortion Motor Waveforms with the MC68HC708MP16"*** *by David Wilson. Note that the MC68HC708MP16 is the predecessor of MC68HC908MRxx Family and contains identical on-chip PWM block.*

Returns:

None

Range Issues:

The dead-time correction algorithm **dtCorrectFull** adds the correction factor to the originally calculated sine wave. It is necessary to ensure that the calculated PWM duty cycles do not exceed the PWM modulus.

Special Issues:

The function **dtCorrectInit** must be called before starting any calls to the function **dtCorrectFull,** to ensure proper functionality.

Design/Implementation:

The **dtCorrectFull** is implemented as a function

Performance:

Table 5-10. dtCorrectFull Performance

Code Size	57 B		
Execution Cycles	Minimum	Maximum	Typical
	96	147	147

Example 8. Dead-Time Correction Algorithm Usage

```
#include "types.h"                  /* Generic SDK types */
#include "dtCorrect.h"              /* dead time correction algorithm */
#include "3ph_acim_dt_correct.h"    /* application header file */


static mc_s3PhaseSystem     pOutputSystem;       /* pointer to output phase
                                                    duty-cycles passed to PWM driver*/
static volatile  SWord16      phase_increment;    /* phase increment between
                                                     the PWM reloads */
static volatile  SWord16      phase_actual;       /* phase A voltage vector position */
static           UByte        amplitude;          /* Amplitude of sinewaves(in % of max.
                                                     phase voltage ampl.) */
static dtCorrect_s            pDtCorrectApp;      /* structure of dead time correction
                                                     algorithm */

#define PWM_DEAD_TIME  0x30   /* PWM Dead-Time Register (defined in appconfig.h) */
#define PWM_PRESC      1      /* PWM prescaller = 1 */


void main (void)
{
   phase_increment = 819;     /* defines sine frequency */
   amplitude = 0x3fff;        /* 50% sine amplitude */
         /* Initialize the dead time correction structure */
```

```
    dtCorrectInit(&pDtCorrectApp);

          /* set full s/w dead time correction */
    IOCTL(PWM, PWM_SET_CURRENT_CORRECTION, PWM_CORRECTION_SOFTWARE);

    EnableInterrupts();       /* Enable ISR */

    while(1);                 /* endless loop */
}

/* ******** PWM RELOAD ISR ******** */

void PwmReloadCallback(void)
{
          /* calculates PWM duty cycle according to the waveform  */

          /* Update phase_actual by phase_increment */
    phase_actual += phase_increment;

          /* Calculate the duty-cycles of for phases A,B,C */
    mcgen3PhWaveSine (amplitude, phase_actual, &pOutputSystem);

          /* Rescale the phase duty-cycles to PWM_MODULO and write to
    value registers for 16kHz PWM */

    PVAL1 = ((UWord16)(pOutputSystem.PhaseA))>>8;
    PVAL3 = ((UWord16)(pOutputSystem.PhaseB))>>8;
    PVAL5 = ((UWord16)(pOutputSystem.PhaseC))>>8;

    /* calculate the modified PVAL values for ead Time Correction */

    PVAL2 = PVAL1 - (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);
    PVAL1 = PVAL1 + (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);
    PVAL4 = PVAL3 - (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);
    PVAL3 = PVAL3 + (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);
    PVAL6 = PVAL5 - (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);
    PVAL5 = PVAL5 + (UWord16)(PWM_DEAD_TIME/2/PWM_PRESC);

  /* The algorithm "dtCorrectFull" needs to be calculated as frequently as
    possible. Ideally the correction should be applied to the individual phases
    immediately when the phase current changes it's magnitude to the low
    values. Frequent calculation of the algorithm limits the effect of time
    delay to the shape of the corrected waveforms */

          /* load actual wave pointer into the dead-time correction structure */
    pDtCorrectApp.pointerA = (SByte)(phase_actual>>8);

          /* load actual DT bits into the dead time correction structure */
    pDtCorrectApp.dtBits = IOCTL (PWM, PWM_GET_CURRENT_SENSING, NULL);

          /* call optimized dead time correction algorithm */
    dtCorrectFull(&pDtCorrectApp);

          /* set bits IPOL1, IPOL2, IPOL3 according to the calculated dead
             time correction output */
    PCTL2 = (PCTL2 & 0xe3)|(pDtCorrectApp.ipolBits);

    IOCTL(PWM, PWM_SET_LOAD_OK, NULL);           /* set LDOK */

}
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

Motorola Japan Ltd.; SPS, Technical Information Center,
3-20-1, Minami-Azabu Minato-ku, Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.;
Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://motorola.com/semiconductors

SDKALUG/D
Rev. 1
11/2002