ACI3-1 System Documentation

Variable Speed Control of 3-Phase AC Induction Motor

Digital Control Systems (DCS) Group

SPRU438
September 2000

TEXAS INSTRUMENTS

Contents

Figures

ACI3–1 Variable Speed Control of Phase AC Induction Motor

The ACI3-1 offers the following:

❑ Open-loop V/Hz control

❑ Space Vector PWM

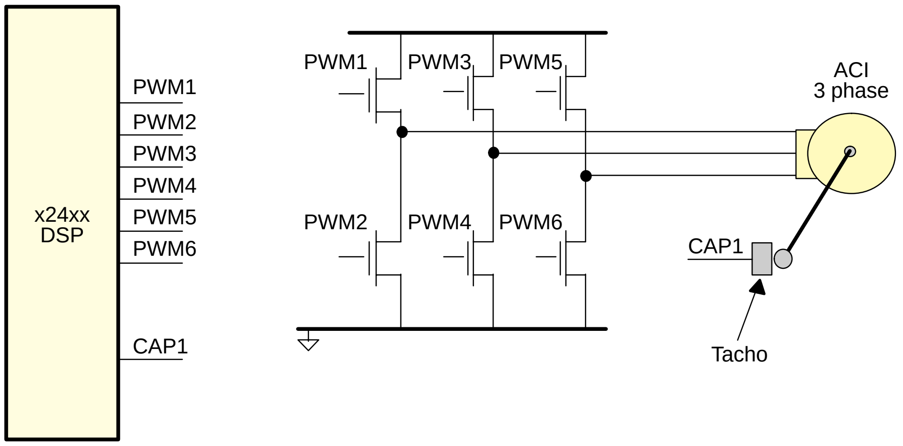
❑ Closed-loop speed control

Topic **Page**

1 System Overview

Figure 1 shows the complete system diagram for a three-phase ACI motor drive. A three phase voltage source inverter is utilized to operate a three pahse ACI motor. Six PWM channels from DSP controller regulates the motor phase voltages by controlling the six power devices. One capture input is utilized to measure the motor speed.

Figure 1. 3-Phase ACI Motor Drive System Diagram



This document describes the 3-phase ACI ac induction motor using TMS320F240™/TMS320F243™/ TMS320F2407™ DSPs. The system implemented here is called ACI3-1 and uses the following software modules:

- ❏ BOX_CAR
- ❏ CAP_EVENT_DRV
- ❏ DATA_LOG
- ❏ FC_PWM_DRV
- ❏ RAMP_CNTL
- ❏ SVGEN_MF
- ❏ SINTB60
- ❏ SPEED_PRD
- ❏ SYS_INIT
- ❏ V_Hz_PROFILE

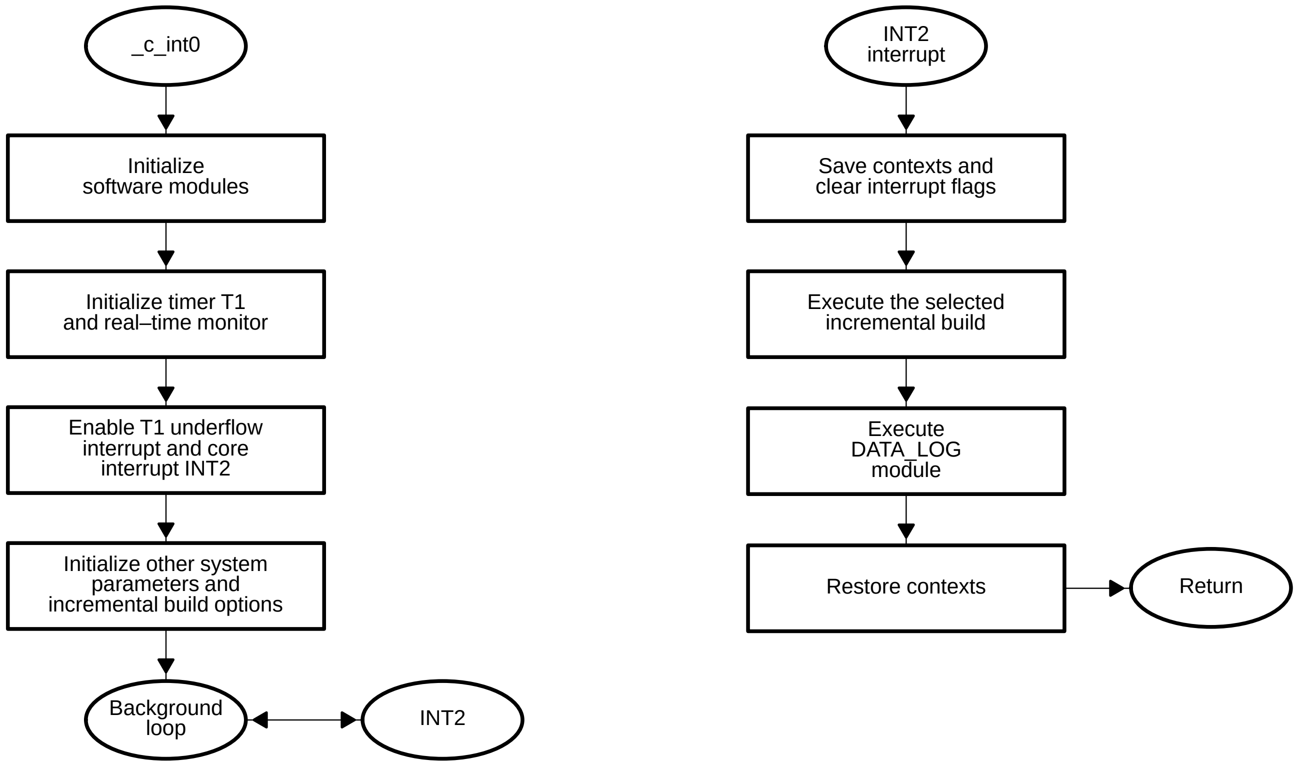
The ACI3-1 System has the following properties:

ASM Program Memory	1700 words
ASM Data Memory	224 words
C Program Memory	1692 words*****
C Data Memory	164 words*****
Development/Emulation	Code Composer 4.1 (or above) with real-time debug
Target controller H/W	Spectrum Digital – F240/F243/F2407 EVMs
Power Inverter H/W	Spectrum Digital – DMC1000/DMC1500
Motor Type	Three-Phase AC Induction (4 pole)

PWM frequency	20 kHz (Timer T1 based)
PWM mode	Symmetrical with Dead band
Interrupts	1 (Timer 1 underflow)
Main sampling loop	20 kHz
Peripheral Resources Used	Timer 1, PWM 1/2/3/4/5/6, 2 ADC Channels

***Note:** The C version of the software at this time excludes the modules DATA_LOG RAMP_CNTL, and BOX_CAR.

Figure 2. Software Flowchart


_c_int0
Initialize software modules
Initialize timer T1 and real–time monitor
Enable T1 underflow interrupt and core interrupt INT2
Initialize other system parameters and incremental build options
Background loop
INT2
INT2 interrupt
Save contexts and clear interrupt flags
Execute the selected incremental build
Execute DATA_LOG module
Restore contexts
Return

2 Hardware and Software Configuration

2.1 Hardware

The experimental system consists of the following hardware components:

- ❑ Spectrum Digital DMC1000 or DMC1500 drive platform;
- ❑ TMS320F240, TMS320F243, or TMS320F2407 EVM platform
- ❑ Spectrum Digital DMC to EVM Interface board;
- ❑ Three Phase AC Induction(ACI) motor
- ❑ IBM compatible development environment including an IBM compatible PC with Code Composer 4.1 or higher installed and XDS510pp emulator.

Refer to the User's Guides and or Manuals for configuration of the above items and their connection to the system. Table 1, Table 2, and Table 3 show some of the other system level choices that must be made.

Table 1. Configuration for DMC1000

Jumper Number	Jumper Setting	Comments
P19–P20	Connect these for ac input neutral connection	
P16–P18	Connect these for full bridge rectifier front-end configuration only	
P16–P17	Connect these for voltage doubler front-end configuration only	Donot use DMC1000/1500 boards in voltage doubler mode with 220VAC input.
JP1, JP2, JP3	Install all jumpers	Use 3-Phase power inverter configuration
JP4	Remove jumper	Use DC bus current sense resistor R5.
JP17	Install jumper on pins 2–3	Use DC bus overcurrent condition to generate PDPINT. Use the potentiometer resistor R19 to adjust the signal Vtrip. This will set the maximum DC bus current.
JP19	Install jumper on pins 1–2	Disable PFC circuit
JP20	Install jumper on pins 2–3	Vccp=Vcc
JP22	Install jumper on pins 1–2	Use s/w to enable/disable DMC1000 PWM signals

Table 2. Configuration for DMC1500

Jumper Number	Jumper Setting	Comments
P21–P22	Connect these for ac input neutral connection	
P15–P17	Connect these for full bridge rectifier front-end configuration only	
P15–P16	Connect these for voltage doubler front-end configuration only	Donot use DMC1000/1500 boards in voltage doubler mode with 220VAC input.
JP1, JP2, JP3	Install all jumpers	Use 3-Phase power inverter configuration
JP4	Remove jumper	Use DC bus current sense resistor R6
JP17	Install jumper on pins 1–2	Disable PFC circuit
JP27	Install jumper on pins 1–2	Use s/w to enable/disable PWM signals on DMC1500

Table 3. Configuration for DMC to EVM Interface Board

Jumper Number	Jumper Setting	Comments
JP1	Remove jumper	
JP2	Install jumper	

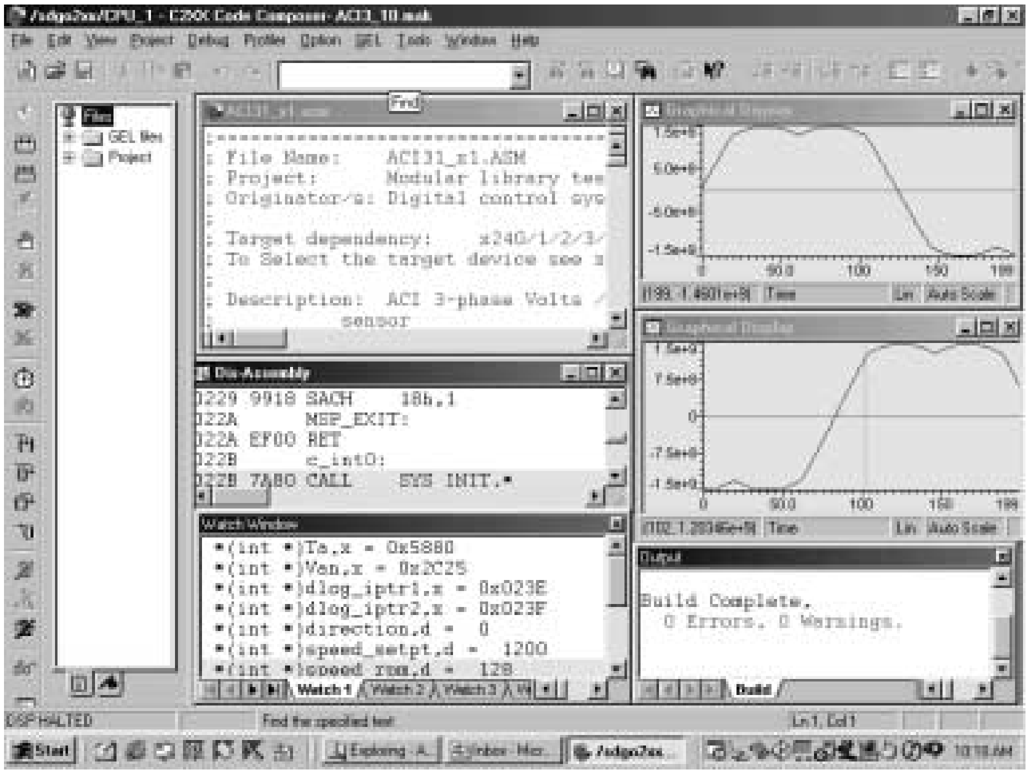
2.2 Loading and Building CC Project for Assembly Case

In order to use the CC workspace file in “C:\TI\DCS\ACI3_1” directory, all the assembly modules in the modular library must be copied into “C:\TI\DCS\ACI3_1” directory. The assembly files needed for this system are – Box_car, Cap_drv, data_log, PID_ID, pwmodrv, rmp_cntl, SINTB60, speed_pr, svgen_mf, sys_init, vhz_prof. The other files in this directory are:

- The ACI system assembly file ACI31_x1.asm;
- The linker command file, ACI31_x1.cmd, that defines memory map and specifies memory allocation;
- Two header files rtvecs.h and x24x_app.h
- CC real time monitor related files c200mnrt.i and c200mnrt.obj
- CC project file ACI3_10.mak
- CC workspace files (aci3_1_phase1, aci3_2_phase2) which contain the setup information for the whole project and debugging environment.

It is also assumed that the emulator used is XDS510pp. Once the directory “C:\TI\DCS\ACI3_1” contains all the necessary files (as mentioned above), the next step is to provide the supply voltage(+5V DC) to the F243/LF2407 EVM and RESET the emulator. Then start the Code Composer and open the project ACI3_10.mak. Now load the workspace file aci3_1_phase1.wks.

Loading the workspace will automatically open up the project file ACI3_10.mak and show all the files relevant to the project. The following shows the expanded project view as part of CC environment aci3_1_phase1wks is loaded:



```
;=====================================
; File Name:     ACI31_x1.ASM
; Project:       Modular library tes
; Originator/s:  Digital control sys
;
; Target dependency:    x240/1/2/3/
; To Select the target device see x
;
; Description:  ACI 3-phase Volts /
;               sensor
```

```
0229 9918 SACH    18h,1
022A      MSP_EXIT:
022A EF00 RET
022B      c_int0:
022B 7A80 CALL    SYS INIT,*
```

```
*(int *)Ta,x = 0x5880
*(int *)Van,x = 0x2C25
*(int *)dlog_iptr1,x = 0x023E
*(int *)dlog_iptr2,x = 0x023F
*(int *)direction,d = 0
*(int *)speed_setpt,d = 1200
*(int *)speed_rpm,d = 128
```

```
Build Complete,
  0 Errors, 0 Warnings.
```

Note that the same project can be built from scratch easily if the workspace file, aci3_1_phase1wks, can't be loaded directly because of differences in CC setup and or emulator used. Refer to CC tutorial for information on a building project.

The variables in the Watch Window can be added manually according to CC tutorial.

From the 'PROGRAM LOAD OPTIONS' in CC select 'Load Program After Build' for automatic loading of the program to the target once the program is compiled.

2.3 Loading and building CC Project for "C" Case

The process for loading and building the C version of the ACI31 system is same as the assembly case with the following exceptions:

❑ **Modules:** The modules DATA_LOG, RMP_CNTL, and BOX_CAR are excluded at this time. The variable outputs that are directed to the DATA_LOG module in ASM are directed to the EVMDAC. The EVMDAC has pointers to the variables that must be set to the variables that are to be output.

For instance suppose a variable ‘`alpha`’, that is in the SVGEN structure `svgen,` is to be output to the DAC Channel 3. The following code is used to set the pointer:

```
dac.qptr3=&vhz.svgen.alpha;
```

(Since the structure `SVGEN` itself is a member of the structure `vhz` , the full variable name is `vhz.svgen.alpha`)

❑ **Incremental Build Control:** The build level is set in build.h in the C infrastructure. This header file is in the aci31\aci31_010\include\ directory.

To set the build level set the BUILDLEVEL symbol to one of the available choices.

```
/*-------------------------------------------------------------
Following is the list of the Build Level choices.
-------------------------------------------------------------*/
#define LEVEL1  1
#define LEVEL2  2
#define LEVEL3  3
/*------------------------------------------------------------
This line sets the BUILDLEVEL to one of the available choices.
------------------------------------------------------------*/
#define   BUILDLEVEL LEVEL3
```

[This is also included in by the system and algorithm source files, and so is visible in the include folder in the Code Composer project window.]

To set the build level to 2, use the following

```
#define   BUILDLEVEL LEVEL2
```

❑ **The target processor selection is in the file TARGET.H.** This file appears in the code composer project as well, or can be found in the directory

```
aci31\aci31_010\include\
```

To change the selection of the processor set the TARGET macro to a different available choice.

```
/*-----------------------------------------------------------
Following is the selection list of the target choices.
Note that the F241 is represented by the F243 and the
LF2407 represents the LF2406 and the LF2402.
-----------------------------------------------------------*/
#define F240   1
#define F243   2
#define F2407  4
#define UNKNOWN 8
/*-------------------------------------------------------------
```

This line sets the target to one of the available choices.

```
------------------------------------------------------------*/
#define TARGET  F243
```

❑ The Object vhz is implemented in the vhz.h. This has many sub–objects such as pid etc. To watch/modify variables such as pid coefficients etc watch

- The entire object vhz, and watch/modify vhz.pid.K0.
- An alternative is to watch vhz.pid and watch/modify pid.K0.

Watching smaller objects in the watch window has the advantage of lowering the amount of data that must be up–loaded to the debug tool.

3 Incremental System Build – Phase 1

Assuming section 1.0 is completed successfully, this section describes the steps for a “minimum” system check-out which confirms operation of system interrupts, some peripheral and target independent modules and one peripheral dependent module.

In the SYSTEM OPTIONS section of the code, select phase 1 incremental build option by setting the constant *phase1_commissioning* to 1. Save the program, compile it and load it to the target.

3.1 Phase 1

- ❑ In this phase, line to neutral voltage waveforms are generated based on the user speed input as shown in Figure 3. Three software modules are utilized in this phase to generate two sine wave forms.
- ❑ Set *phase1_commissioning = 1* in *ACI31_x1.asm* as shown below:

```
phase1_commissioning        .set      1
phase2_commissioning        .set      0
phase3_commissioning        .set      0
```

- ❑ The software modules utilized in this phase are – RAMP_CNTL, V_Hz_PROFILE, SVGEN_MF
- ❑ RAMP_CNTL is utilized to avoid sudden change in speed command.
- ❑ V_Hz_PROFILE generates a voltage command “v_out”. V_Hz_PROFILE also maintains a fixed ratio between the input frequency and voltage command.
- ❑ The module SVGEN_MF generates three line-to-neutral waveforms. These waveforms are generated by space vector PWM technique and contain fundamental frequency as well as a third harmonic component to maximize the utilization DC bus voltage.
- ❑ Although line to neutral waveform contains third harmonic, the actual line to line voltage seen by the load will not have any third harmonic component. This line to line wave form is also generated by taking the appropriate difference between the line to neutral voltages. For example, line to line wave form between phases A and B can be calculated as Van – Vbn, where, Van = waveform between phase A and neutral and Vbn = waveform between phase B and neutral.
- ❑ The generated waveforms can be watched using the software module “DATA_LOG”.
- ❑ In order to watch line to neutral waveforms in real time initialize *dlog_iptr1 = Ta* (line to neutral waveform for phase A) and *dlog_iptr2 = Tb* (line to neutral waveform for phase A) (Figure 4)
- ❑ In order to watch line to neutral and line to line waveform in real-time initialize *dlog_iptr1 = Ta* and *dlog_iptr2 = Vab* (Figure 5)
- ❑ Figure 12 shows the complete software block diagram of Phase 1.

Figure 3. Phase 1 Incremental Build for ACI1-1

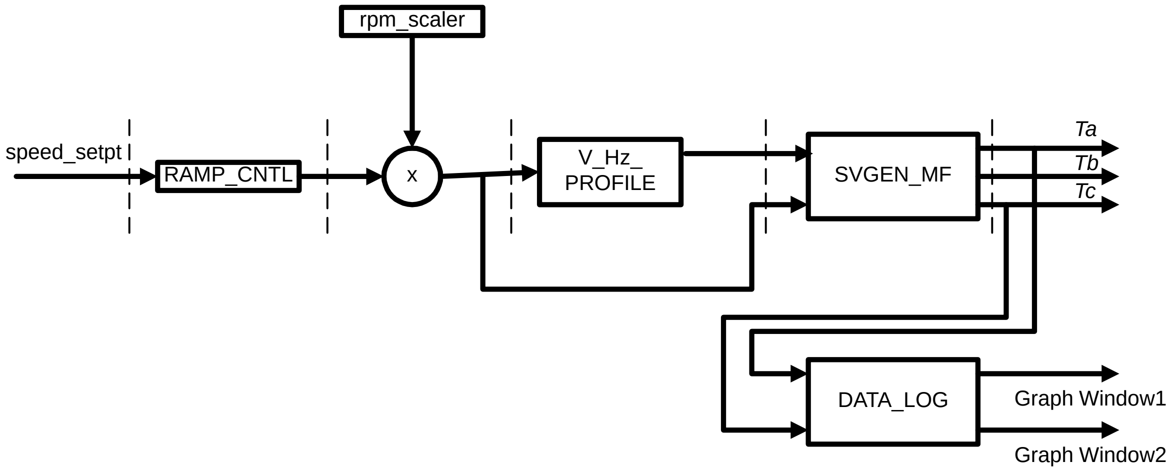
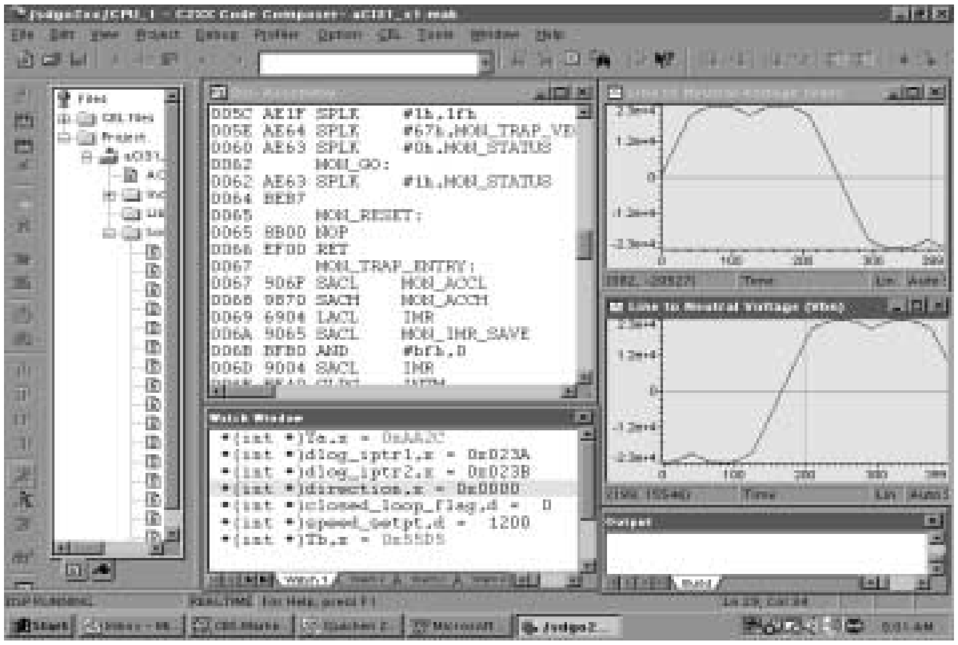


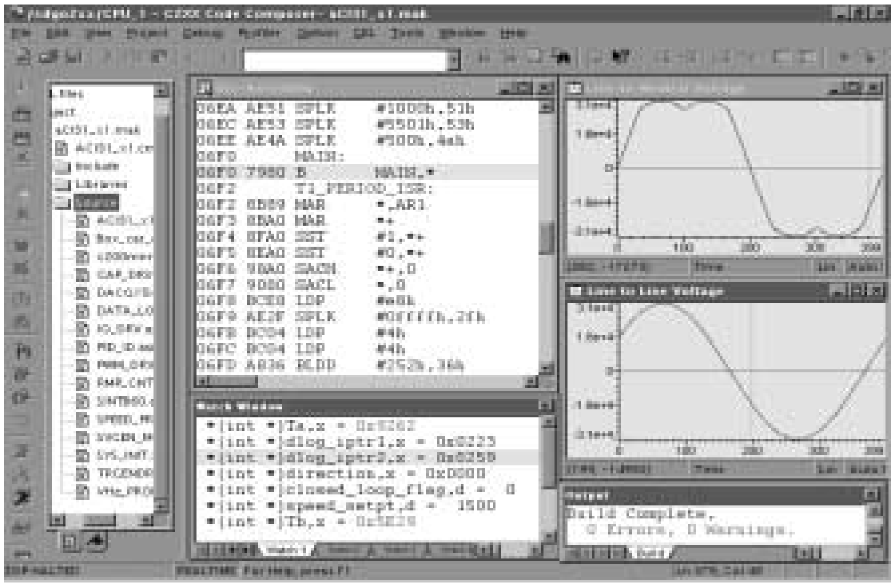
Figure 4. CC Window for Phase 1 Incremental Build of ACI3-1 Showing Line to Neutral Waveforms


C2XX Code Composer - aci31_s1.mak
File Edit View Project Debug Profiler Option GEL Tools Window Help
Files
GEL files
Project
Line to Neutral Voltage (Van)
Line to Neutral Voltage (Vbn)
Watch Window
Watch 1
Output
Time
Lin
Auto
DSP RUNNING
REALTIME For Help, press F1
Start

```
005C AE1F SPLK     #1h,1fh
005E AE64 SPLK     #67h,MON_TRAP_VE
0060 AE63 SPLK     #0h,MON_STATUS
0062           MON_GO:
0062 AE63 SPLK     #1h,MON_STATUS
0064 BEB7
0065           MON_RESET:
0065 8B00 NOP
0066 EF00 RET
0067           MON_TRAP_ENTRY:
0067 906F SACL     MON_ACCL
0068 9870 SACH     MON_ACCH
0069 6904 LACL     IMR
006A 9065 SACL     MON_IMR_SAVE
006B BFB0 AND      #bfh,0
006D 9004 SACL     IMR
```

```
*(int *)Ta.x = 0xAA2C
*(int *)dlog_iptr1.x = 0x023A
*(int *)dlog_iptr2.x = 0x023B
*(int *)direction.x = 0x0000
*(int *)closed_loop_flag.d =  0
*(int *)speed_setpt.d =  1200
*(int *)Tb.x = 0x55D5
```

Figure 5. CC Window for Phase 1 Incremental Build of ACI3-1 Showing Line to Neutral (Ta or Van) and Line-to-line Waveform (Vab)



```
06EA AE51 SPLK     #1000h,51h
06EC AE53 SPLK     #5501h,53h
06EE AE4A SPLK     #500h,4ah
06F0      MAIN:
06F0 7980 B        MAIN,*
06F2      T1_PERIOD_ISR:
06F2 8B89 MAR      *,AR1
06F3 8BA0 MAR      *+
06F4 8FA0 SST      #1,*+
06F5 8EA0 SST      #0,*+
06F6 98A0 SACH     *+,0
06F7 9080 SACL     *,0
06F8 BCE8 LDP      #e8h
06F9 AE2F SPLK     #0ffffh,2fh
06FB BC04 LDP      #4h
06FC BC04 LDP      #4h
06FD A836 BLDD     #252h,36h
```

```
*(int *)Ta.x = 0x8262
*(int *)dlog_iptr1.x = 0x0223
*(int *)dlog_iptr2.x = 0x0250
*(int *)direction.x = 0x0000
*(int *)closed_loop_flag.d =  0
*(int *)speed_setpt.d =  1500
*(int *)Tb.x = 0x5E28
```

```
Build Complete,
  0 Errors, 0 Warnings.
```

4 Incremental System Build – Phase 2

Successful completion of this phase will enable one to operate a three-phase AC induction motor with open loop speed control.

In Phase 2 of incremental system build, a three phase AC induction motor will be operated using DMC1000 board (www.spectrumdigital.com) and EVM board from TI. In Phase 2a little software checks are made and then in Phase 2b an inverter and motor is used to operate a three phase ac induction motor with variable speed control.

4.1 Incremental System Build – Phase 2a

❑ Set *phase2_commissioning = 1* in *ACI31_x1.asm* as shown below:

```
phase1_commissioning     .set    0
phase2_commissioning     .set    1
phase3_commissioning     .set    0
```

❑ The software block diagram of Phase 2a is shown in Figure 6.

❑ The additional software module added in this Phase 2a are FC_PWM_DRV.

❑ FC_PWM_DRV configures PWM module of the DSP controller. It also provides the required dead band through avoid shoot thorough faults.

❑ The variable “direction” can be set to “1” or “0” to change the direction of rotation of the motor. When direction is equal to “1’, The phase sequence is a-b-c. If direction is equal to “0” than the opposite will happen. The available graph display in CC can be utilized to observe this. Put dlog_iptr1 = Mfunc_c1 and dlog_iptr2 = Mfunc_c2. Figure 7 and Figure 8 show how phase to neutral waveforms lead or lag each other depending on the value of “direction”.

Figure 6. Phase 2a Software Block Diagram for ACI3-1 System

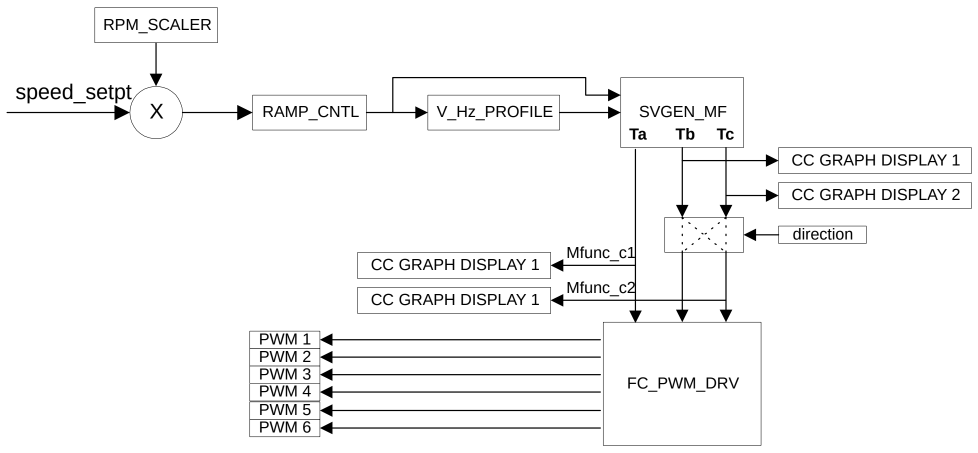
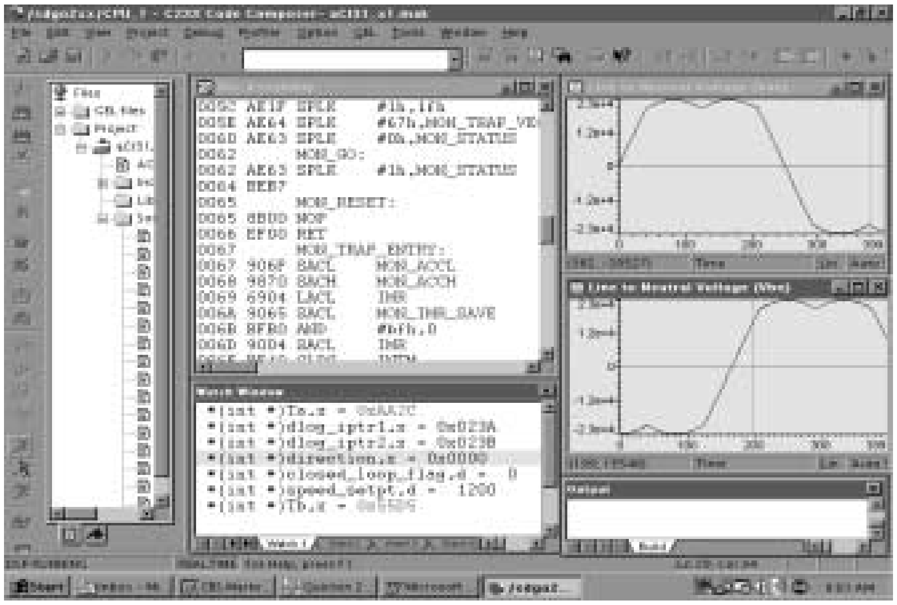


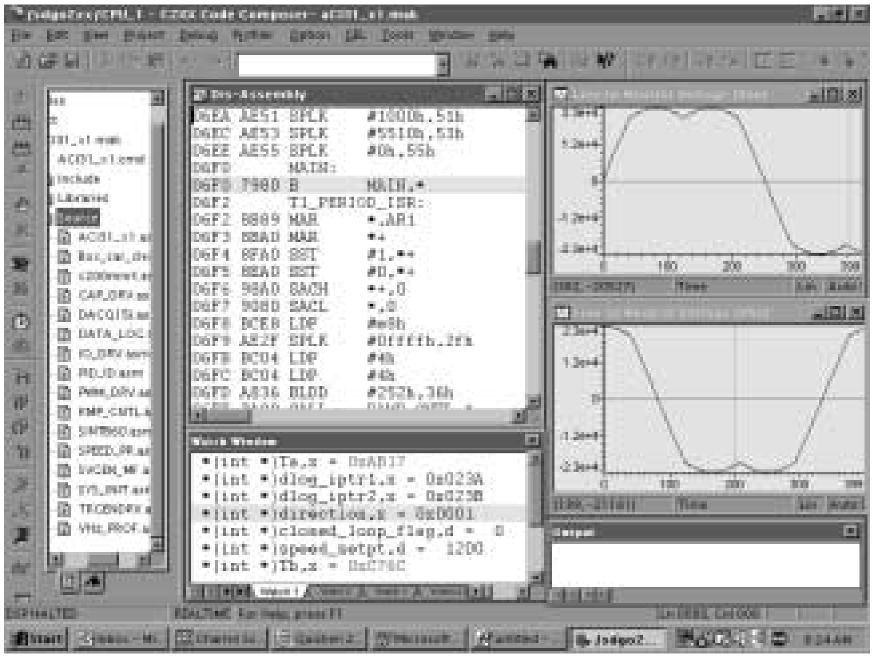
Figure 7. CC Display Window With dlog_iptr1=Mfunc_c1, dlog_iptr2=Mfunc_c2, direction=0 (wave 1 leads wave 2)



```
005C AE1F SPLK    #1h,1fh
005E AE64 SPLK    #67h,MON_TRAP_VE
0060 AE63 SPLK    #0h,MON_STATUS
0062      MON_GO:
0062 AE63 SPLK    #1h,MON_STATUS
0064 BEB7
0065      MON_RESET:
0065 8B00 NOP
0066 EF00 RET
0067      MON_TRAP_ENTRY:
0067 906F SACL    MON_ACCL
0068 9870 SACH    MON_ACCH
0069 6904 LACL    IMR
006A 9065 SACL    MON_IMR_SAVE
006B BFB0 AND     #bfh,0
006D 9004 SACL    IMR
```

```
*(int *)Ta.x = 0xAA2C
*(int *)dlog_iptr1.x = 0x023A
*(int *)dlog_iptr2.x = 0x023B
*(int *)direction.x = 0x0000
*(int *)closed_loop_flag.d =  0
*(int *)speed_setpt.d =  1200
*(int *)Tb.x = 0x55D5
```

Figure 8. CC Display Window With dlog_iptr1=Mfunc_c1, dlog_iptr2=Mfunc_c2, direction=1 (wave 1 lags wave 2)



```
06EA AE51 SPLK    #1000h,51h
06EC AE53 SPLK    #5510h,53h
06EE AE55 SPLK    #0h,55h
06F0      MAIN:
06F0 7980 B       MAIN,*
06F2      T1_PERIOD_ISR:
06F2 8B89 MAR     *,AR1
06F3 8BAD MAR     *+
06F4 8FA0 SST     #1,*+
06F5 8EAD SST     #0,*+
06F6 98A0 SACH    *+,0
06F7 9080 SACL    *,0
06F8 BCE8 LDP     #e8h
06F9 AE2F SPLK    #0ffffh,2fh
06FB BC04 LDP     #4h
06FC BC04 LDP     #4h
06FD A836 BLDD    #252h,36h
```

```
*(int *)Ta.x = 0xAB17
*(int *)dlog_iptr1.x = 0x023A
*(int *)dlog_iptr2.x = 0x023B
*(int *)direction.x = 0x0001
*(int *)closed_loop_flag.d =  0
*(int *)speed_setpt.d =  1200
*(int *)Tb.x = 0xC78C
```

Note: Motor rotational direction in Figure 8 will be opposite of rotational direction in Figure 7.

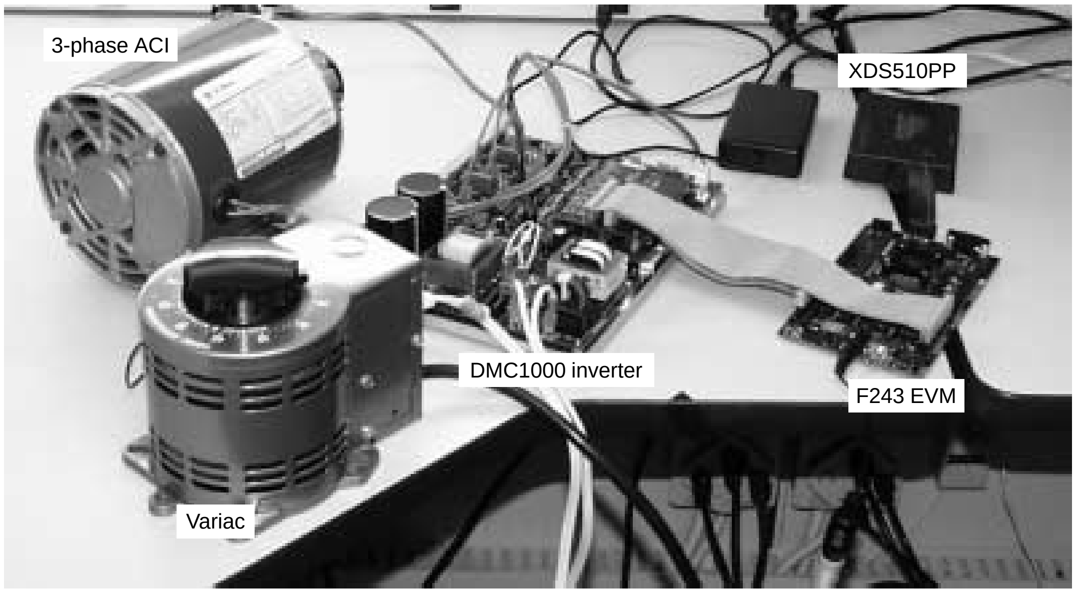
4.2 Incremental System Build – Phase 2b

In this phase a three phase AC induction motor is connected with a voltage source inverter and the above mentioned software is utilized to implement a variable speed drive. A 243EVM is interfaced with DMC1000 board to setup the controller and power stage of the drive. The following steps are needed for proper setup:

- ❑ P1 (I/O) terminal of 243EVM connects to P4 (I/O) terminal of DMC to EVM interface board (use the supplied ribbon cable).
- ❑ P2 (Analog) terminal of 243EVM connects to P3 (I/O) terminal of DMC to EVM interface board (use the supplied ribbon cable).
- ❑ JTAG connector of XDS510PP connects to the JTAG connector available in 243EVM board.
- ❑ DMC1000 board is supplied 24VDC in P15.
- ❑ 243EVM is supplied 5VDC in J1.
- ❑ DMC1000 is supplied from a variac between LINE(P24) and NEUTRAL (P25).
- ❑ P19 and P20 are connected on DMC1000.
- ❑ P16 and P17 are connected on DMC1000.
- ❑ JP19 on DMC1000 should be between 1 and 2 (disables power factor correction circuit).
- ❑ Motor phase 1 connects to P2 on DMC1000.
- ❑ Motor phase 2 connects to P3 on DMC1000.
- ❑ Motor phase 3 connects to P5 on DMC1000.
- ❑ Turn the variac all the CCW to zero voltage position.
- ❑ Connect the variac to 120V wall socket.
- ❑ Connect a current probe with any motor phase.
- ❑ Power ON EVM and DMC1000 boards.
- ❑ Start CC and load the software.
- ❑ Once the software is running, apply DC bus voltage by rotating the variac. At this time the current probe should show sinusoidal phase current and after sufficient DC bus voltage, the motor will start rotating.
- ❑ The direction of rotation will depend on two variables – First the motor phase connections and second the value of the variable “direction”.
- ❑ Changing the value of “direction” as mentioned in the previous section (Phase 2a) can change the rotational direction.
- ❑ Modifying the variable “speed_setpt” can vary the motor speed. The maximum speed is fixed at 2500RPM. However, user can change that by modifying V_Hz_PROFILE software module.

- ❑ Figure 9 shows the hardware setup.
- ❑ Figure 13 shows the complete software block diagram of Phase 2.

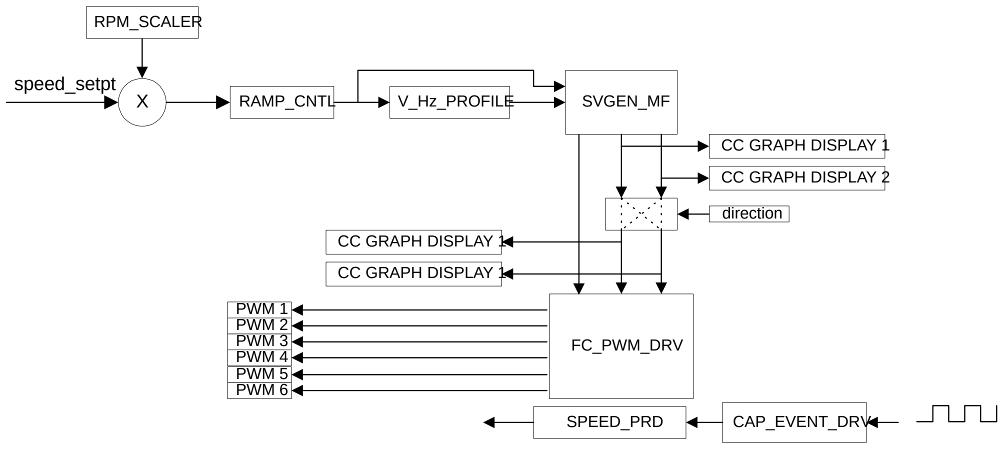
Figure 9. Complete Hardware Setup for Phase 2b Implementation



5 Incremental System Build – Phase 3

Successful completion of this phase will enable one to operate a three-phase AC induction motor with closed loop speed control. Closed loop speed control is not implemented in Phase 4.0a, however, software block is utilized to measure the speed. This will verify the proper operation of speed measuring hardware (sprocket and hall effect sensor). In phase 3.0b, the speed loop is closed. Figure 10 shows the software block diagram for phase 3.0a.

Figure 10. Software Block Diagram for Phase 3a



5.1 Incremental System Build – Phase 3a

In this phase the same three phase AC induction motor is operated in variable speed mode. However, in this phase, the speed of the motor is measured using a hall effect sensor and hardware capture module of the DSP controller. In order to measure speed the following assembly files are utilized:

❑ CAP_DRV.asm	: configures the capture module
❑ SPEED_PR.asm	: calculates speed from captured values
❑ Box_car	: calculates the average of the calculated speed.
	: This eliminates jitters in speed sensing hardware.

The following steps will ensure proper operation of Phase 3a:

❑ Set *phase3_commissioning = 1* in *ACI31_x1.asm* as shown below:

```
phase1_commissioning    .set    0
phase2_commissioning    .set    0
phase3_commissioning    .set    1
```

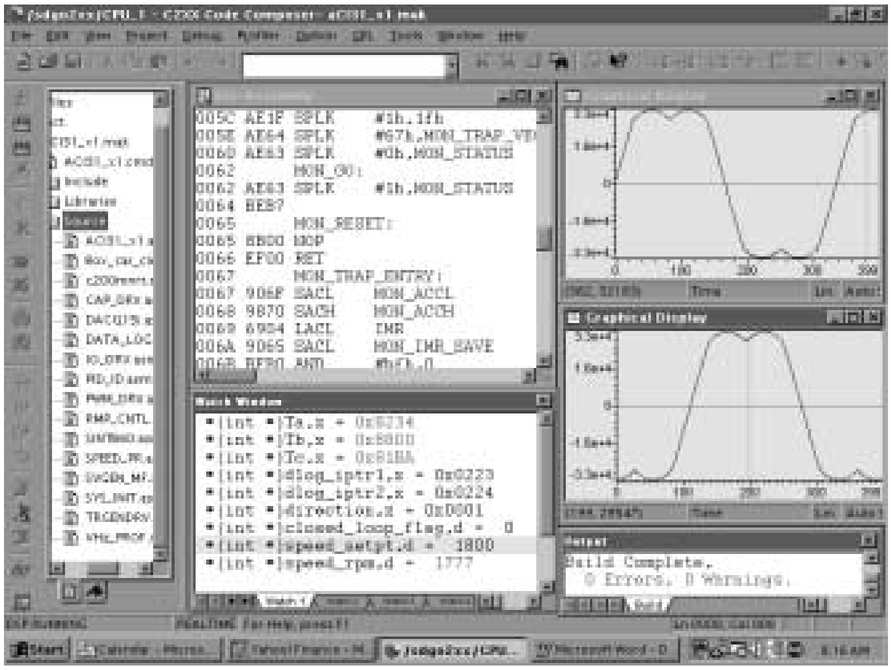
❑ The Hall effect speed sensor output is connected to P27 of DMC1000. P27 has ground, power and capture input. The available ground and power from P27 can be utilized as sensor ground and power. The voltage output at P27 can be set at various level to match the sensor requirement.

❑ The remaining hardware setup is similar to Phase 2b.

❑ After starting the software, the motor will start rotating as the DC bus voltage is applied by rotating the variac from zero to higher voltages.

- ❑ The measured speed can be watched by adding the variable “speed_rpm” in the watch window.
- ❑ Figure 11 shows CC window when the motor is operating in Phase 3a. The following observations can be made from Figure 11.
- ❑ Line to neutral voltages contain fundamental and third harmonics
- ❑ Speed command, speed_setpt = 1800RPM.
- ❑ Measured speed, speed_rpm = 1777RPM. Since this is an induction motor, the measured speed equals to command speed minus the slip speed.
- ❑ Figure 14 shows the complete software block diagram of Phase 3.

Figure 11. CC Window for Phase 3a – Commanded Speed is 1800RPM, Open-loop speed is 1777RPM



```
005C AE1F SPLK     #1h,1fh
005E AE64 SPLK     #67h,MON_TRAP_VE
0060 AE63 SPLK     #0h,MON_STATUS
0062      MON_GO:
0062 AE63 SPLK     #1h,MON_STATUS
0064 BEB7
0065      MON_RESET:
0065 8B00 NOP
0066 EF00 RET
0067      MON_TRAP_ENTRY:
0067 906F SACL     MON_ACCL
0068 9870 SACH     MON_ACCH
0069 6904 LACL     IMR
006A 9065 SACL     MON_IMR_SAVE
006B BFB0 AND      #0fh,0
```

Watch Window

```
*(int *)Ta.x = 0x0234
*(int *)Tb.x = 0x8000
*(int *)Tc.x = 0x81BA
*(int *)dlog_iptr1.x = 0x0223
*(int *)dlog_iptr2.x = 0x0224
*(int *)direction.x = 0x0001
*(int *)closed_loop_flag.d =  0
*(int *)speed_setpt.d =  1800
*(int *)speed_rpm.d =  1777
```

Output

```
Build Complete,
  0 Errors, 0 Warnings.
```

Figure 12. Software Block Diagram of Phase–1

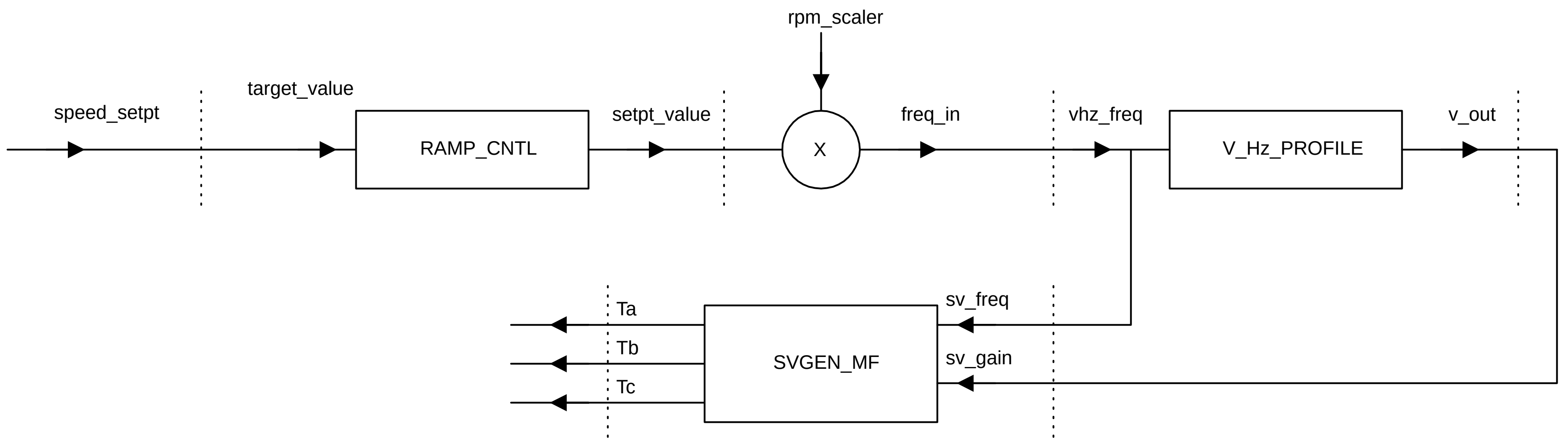


Figure 13. Software Block Diagram of Phase–2

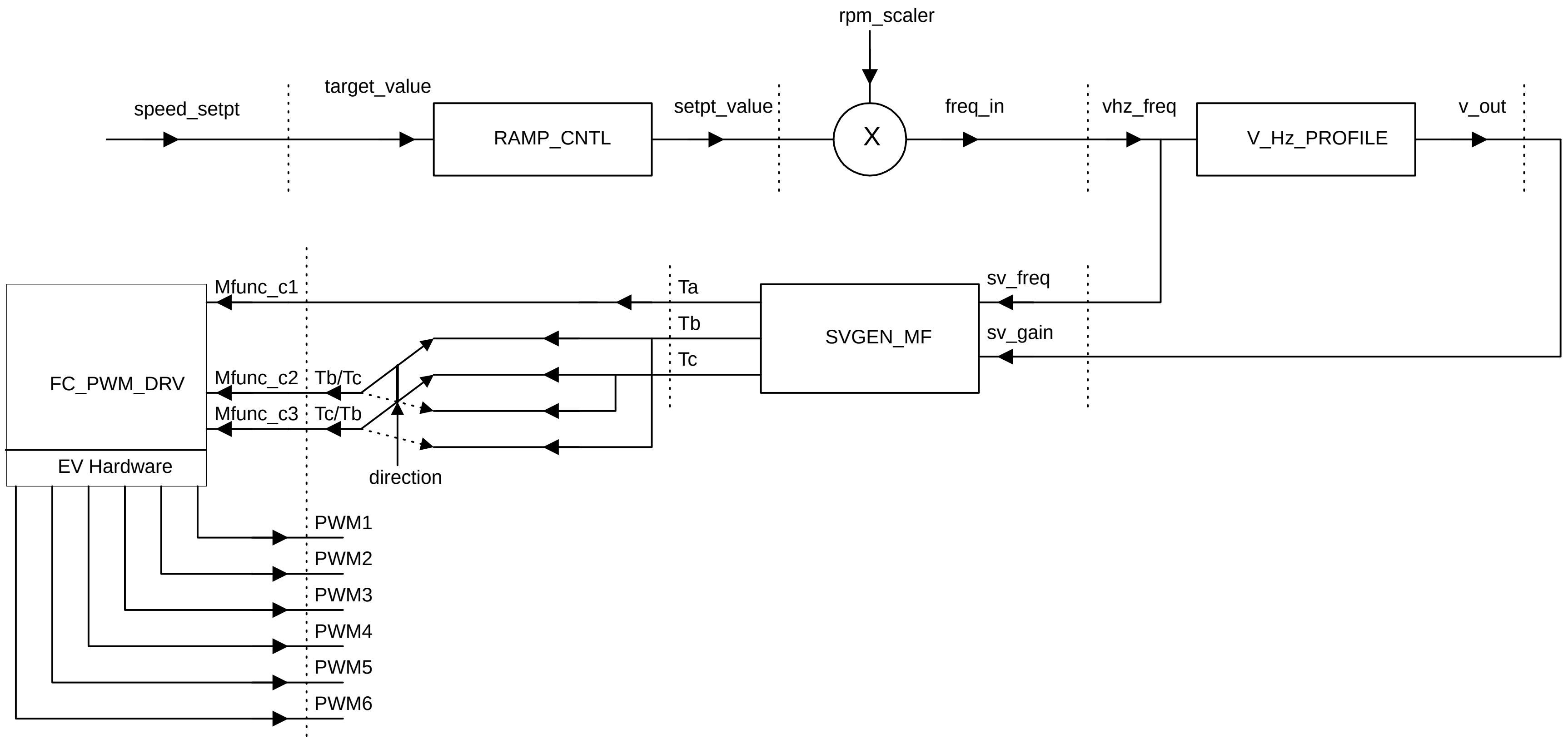
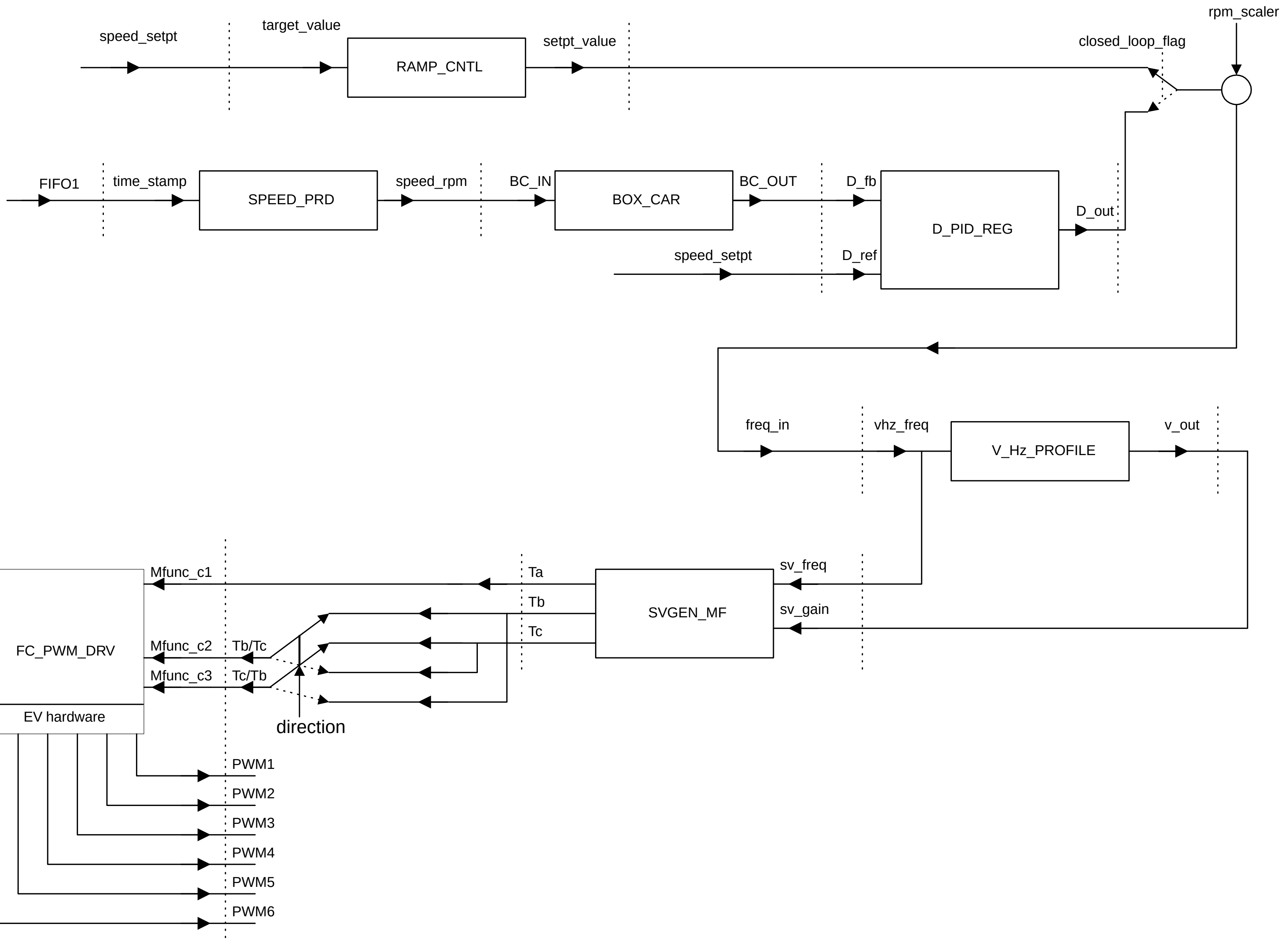


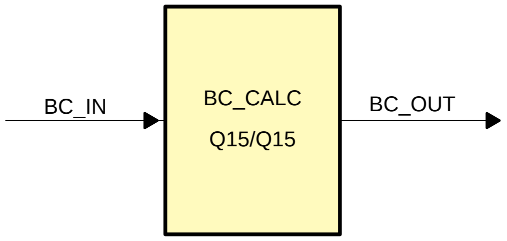
Figure 14. Software Block Diagram of Phase–3



BC_CALC — *Averaging Box Car*

Description

This software module calculates the average value of a s/w variable. The output can be rescaled and the size of buffer used for storing the averaging data is selectable.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Asembly File Name: box_car.asm

ASM Routines: BC_CALC, BC_INIT

C-Callable ASM File Names: box_car.asm, box_car.h

Item	ASM Only	C-Callable ASM	Comments
Code size	47 words	46 words‡	
Data RAM	69† words	69† words‡	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† For 64-word buffer size.
‡ Each pre-initialized BOXCAR structure occupies (5+BC_SIZE) words in the data memory and (7+BC_SIZE) words in the .cinit section.

Direct ASM Interface

Table 1. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	BC_IN	Input to be averaged	Q15	–1 –> 0.999
Output	BC_OUT	Averaged output with the selectable buffer size	Q15	–1 –> 0.999
Init / Config	BC_SIZE	The buffer size	Q0	2, 4, 8, 16, …
	bc_scaler	The scaling factor	Q15	–1 –> 0.999

Routine names and calling limitation:
There are two routines involved:

BC_CALC, the main routine; and
BC_INIT, the initialization routine.

The initialization routine must be called during program initialization. The BC_CALC routine must be called in the control loop.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   BC_INIT, BC_CALC      ;function call
.ref   BC_IN, BC_OUT       ;Inputs/Outputs
```

Memory map:
All variables are mapped to an uninitialized named section, bc, which can be allocated to any one data page. However, the buffer data is mapped to an uninitialized named section, farmem.

Example:
In the interrupt service routine call the module and read results as follows:

```
LDP  #BC_IN                  ; Set DP for module inputs
BLDD #input_var1,BC_IN       ; Pass input variables to module inputs

CALL BC_CALC

LDP  #output_var1              ; Set DP for module output
BLDD #BC_OUT, output_var1      ; Pass output to other variables
```

C/C-Callable ASM Interface

Object Definition

The structure of the BOXCAR object is defined in the header file, box_car.h, as seen in the following:

```
#define    BC_SIZE  64

  typedef struct  { int  BC_IN;               /* Input: Box–Car input (Q15) */
                    int  BC_PTR;              /* Variable: Box–car buffer pointer */
                    int  BC_BUFFER[BC_SIZE];  /* Variable: Box–car buffer (Q15) */
                    int  BC_OUT;              /* Output: Box–car output (Q15) */
                    int  bc_scaler;           /* Parameter: Box–car scaler (Q15) */
                    int  (*calc)();           /* Pointer to calculation function */
                  } BOXCAR;
```

Special Constants and Datatypes

BOXCAR
The module definition itself is created as a data type. This makes it convenient to instance a BOXCAR object. To create multiple instances of the module simply declare variables of type BOXCAR.

BOXCAR_DEFAULTS
Initializer for the BOXCAR object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, box_car.h.

Methods

void calc(BOXCAR *);
This default definition of the object implements just one method – the runtime compute function for averaging. This is implemented by means of a function pointer, and the default initializer sets this to bc_calc function. The argument to this function is the address of the BOXCAR object. Again, this statement is written in the header file, box_car.h.

Module Usage

Instantiation:
The following example instances two such objects:

```
BOXCAR bc1, bc2;
```

Initialization:
To instance a pre-initialized object:

```
BOXCAR bc1 = BOXCAR_DEFAULTS;
BOXCAR bc2 = BOXCAR_DEFAULTS;
```

Invoking the compute function:

```
bc1.calc(&bc1);
bc2.calc(&bc2);
```

Example:

Lets instance two BOXCAR objects, otherwise identical, and compute the averaging values of two different s/w variables. The following example is the c source code for the system file.

```
BOXCAR bc1= BOXCAR_DEFAULTS;      /* instance the first object */
BOXCAR bc2= BOXCAR_DEFAULTS;      /* instance the second object */

main()
{

    bc1.BC_IN = input1;              /* Pass inputs to bc1 */
    bc2.BC_IN = input2;              /* Pass inputs to bc2 */

}

void interrupt periodic_interrupt_isr()
{

    bc1.calc(&bc1);            /* Call compute function for bc1 */
    bc2.calc(&bc2);            /* Call compute function for bc2 */

    output1 = bc1.BC_OUT;      /* Access the outputs of bc1 */
    output2 = bc2.BC_OUT;      /* Access the outputs of bc2 */

}
```

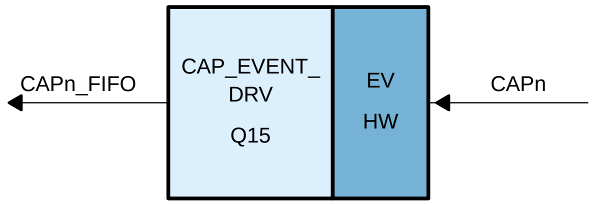
Background Information

This s/w module computes the average of the runtime values of the selected input variable. The size of the buffer used to keep the data is selectable with the power of two, i.e., 2, 4, 8, 16, 32, 64, …. The default buffer size is 64. For different buffer size modify the code (valid for both ASM and CcA versions) as required. The following instruction is added or deleted, according to the buffer size, at the location indicated in the code. This divides the number in accumulator by two.

```
SFR                     ; Number of times SFR need to be executed
                        ; is, log2(BC_SIZE)
```

Description

This module provides the instantaneous value of the selected time base (GP Timer) captured on the occurrence of an event. Such events can be any specified transition of a signal applied at the event manager (EV) capture input pins of 24x/24xx devices.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: cap_drv.asm

C-Callable Version File Names: F243CAP.h, F243CAPx.c, F2407CAPx.c, F2407CAP.H, CAPTURE.H

Item	ASM Only	C-Callable ASM	Comments
Code size	32 words	54 words (49 words .text, 5 words .cinit)	
Data RAM	1 words	6 words	
Multiple instances	No	Yes†	Multiple instances must be initialized to point to different capture pin routines.

† Creating multiple instances pointing to the same capture pin can cause undefined results.

Direct ASM Interface

Table 2. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	CAPn (n=1,2,3,4)	Capture input signals to 24x/24xx device	N/A	N/A
Outputs	CAPnFIFO (n=1,2,3,4)	Capture unit FIFO registers.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.	N/A	N/A
	CLK_prescaler_bits	Initialize this clock prescaler variable. The default value is set to 4. To use this value call the CAP_EVENT_DRV_INIT routine only. For a different value modify this variable and also call the other initialization routine CAP_EVENT_DRV_CLKPS_INIT. The correct value for this parameter is calculated in the Excel file with the user input of the desired clock prescaler (1,2,4,8,16,32,64,128).	Q0	0–7

Variable Declaration:
In the system file include the following statements:

```
.ref   CAP_EVENT_DRV, CAP_EVENT_DRV _INIT    ;function call

.ref   CAP_EVENT_DRV_CLKPS_INIT              ;function call

.ref   CLK_prescaler_bits                    ;parameter
```

Memory map:
Not required.

Example:

```
CALL CAP_EVENT_DRV_INIT
ldp  #CLK_prescaler_bits
splk #7, CLK_prescaler_bits   ;To specify a prescaler of 128
CALL CAP_EVENT_DRV_CLKPS_INIT

ldp  #output_var1          ;Set DP for output variable
bldd #CAP1FIFO,output_var1 ;Pass module o/ps to output vars
bldd # CAP2FIFO, output_var2
bldd # CAP3FIFO, output_var3
```

C/C-Callable ASM Interface

Object Definition

The structure of the CAPTURE object is defined by the following struct

```
/*-------------------------------------------------------------------
Define the structure of the Capture Driver Object
-------------------------------------------------------------------*/
typedef struct { int time_stamp;
                 int (*init)(); /*Pointer to the init function */
                 int (*read)(); /*Pointer to the init function */
               } CAPTURE;
```

Table 3. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Input Pins	–	–		Inputs are logic levels on hardware pins.
Output	Time_stamp	An Integer value read from timer assigned to the capture unit.	Q0	–32768 to 32767

Special Constants and Datatypes

CAPTURE
The module definition itself is created as a data type. This makes it convenient to instance an interface to the CAPTURE pin(s).

CAPTURE_DEFAULTS
Initializer for the CAPTURE Object. This provides the initial values to the terminal variables as well as method pointers.

CAPTURE_handle
This is typedef'ed to CAPTURE *.

Methods

void init(CAPTURE_handle)
Initializes the CAPTURE unit on the device to activate the capture function.

int read(CAPTURE_handle)
Reads a time stamp value from the timer associated with the capture unit. Note that the time stamp is placed in the capture object. The return value of the function is either 0 or 1. If the function read a value from the hardware, i.e. if a capture event has occurred, then the function returns 0. Otherwise the return value is 1.

Module Usage

Instantiation:
The interface to the Capture unit on the device is instanced thus:

```
CAPTURE cap1;
```

Initialization:
To instance a pre-initialized object

```
CAPTURE cap1=CAP1_DEFAULTS;
```

Invoking the initialization function:

```
cap1.init(&cap1);
```

Reading a time stamp from the capture unit:

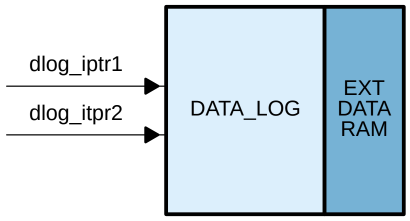
```
cap1.read(&cap1);
```

Example:
Lets instance one CAPTURE object, init it and invoke the read function to fetch a time stamp.

```
CAPTURE cap1 CAP1_DEFAULTS; /*Instance the Capture interface object    */

main()
{

     cap1.init(&cap1);

}
void interrupt periodic_interrupt_isr()
{

     int status;
     int time_of_event;

     status=cap1.read(&cap1);

     /* if status==1 then a time stamp was not read,
        if status==0 then a time stamp was read.

     if(status==0)
     {
     time_of_event=cap1.time_stamp;
     }
}
```

Description

This module stores the realtime values of two user selectable s/w variables in the external data RAM provided on the 24x/24xx EVM. Two s/w variables are selected by configuring two module inputs, *dlog_iptr1* and *dlog_iptr2*, point to the address of the two variables. The starting addresses of the two RAM locations, where the data values are stored, are set to 8000h and 8400h. Each section allows logging of 400 data values.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: x24x/x24xx

Assembly File Name: data_log.asm

ASM Routines: DATA_LOG, DATA_LOG_INIT

C-Callable ASM File Names: data_log1.c, data_log2.asm, data_log.h

Item	ASM Only	C-Callable ASM	Comments
Code size	80 words	118 words†	
Data RAM	8 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized DATALOG structure instance consumes 14 words in the data memory and 16 words in the .cinit section.

Direct ASM Interface

Table 4. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	dlog_iptrx (x=1,2)	These inputs contain the addresses of the desired variables.	N/A	N/A
Outputs	none			
Init / Config	dlog_iptrx (x=1,2)	Initialize these inputs with the addresses of the desired variables. However, this initialization is optional, since these input variables can also be loaded with the addresses of any s/w variables from the Code Composer watch window.		

Variable Declaration:
In the system file include the following statements:

```
.ref   DATA_LOG, DATA_LOG _INIT       ;function call
.ref   dlog_iptr1, dlog_iptr2         ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘data_log’

Example:
During the initialization part of the user code, initialize the module inputs with the address of the desired variables as shown below:

```
CALL DATA_LOG_INIT   ;Initializes DAC parameters

ldp  #dlog_iptr1              ;Set DP for module inputs
splk #input_var1, dlog_iptr1  ;Pass input variables to module inputs
splk #input_var2, dlog_iptr2
```

Then in the interrupt routine just call the module to store the values of the intended variables in the external RAM.

```
CALL DATA_LOG
```

Note:

This module does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing.

C/C-Callable ASM Interface

Object Definition

The structure of the DATALOG object is defined in the header file, data_log.h, as shown in the following:

```
typedef struct  { int  *dlog_iptr1;     /* Input: First input pointer (Q15) */
                  int  *dlog_iptr2;     /* Input: Second input pointer (Q15) */
                  int  trig_value;      /* Input: Trigger point (Q15) */
                  int  graph_ptr1;      /* Variable: First graph address */
                  int  graph_ptr2;      /* Variable: Second graph address */
                  int  dlog_skip_cntr;  /* Variable: Data log skip counter */
                  int  dlog_cntr;       /* Variable: Data log counter */
                  int  task_ptr;        /* Variable: Task address */
                  int  dlog_prescale;   /* Parameter: Data log prescale */
                  int  dlog_cntr_max;   /* Parameter: Maximum data buffer */
                  int  dl_buffer1_adr;  /* Parameter: Buffer starting address 1 */
                  int  dl_buffer2_adr;  /* Parameter: Buffer starting address 2 */
                  int  (*init)();       /* Pointer to init function */
                  int  (*update)();     /* Pointer to update function */
                } DATALOG;
```

Special Constants and Datatypes

DATALOG
The module definition itself is created as a data type. This makes it convenient to instance a DATALOG object. To create multiple instances of the module simply declare variables of type DATALOG.

DATALOG_DEFAULTS
Initializer for the DATALOG object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, data_log.h.

Methods

void init(DATALOG *);
void update(DATALOG *);
This default definition of the object implements two methods – the initialization and run-time update function for data logging. This is implemented by means of a function pointer, and the default initializer sets these to data_log_init and data_log_update functions. The argument to these functions is the address of the DATALOG object. Again, this statement is written in the header file, data_log.h.

Module Usage

Instantiation:
The following example instances two such objects:

```
DATALOG  dlog1, dlog2;
```

Initialization:
To instance a pre-initialized object:

```
DATALOG dlog1 = DATALOG_DEFAULTS;
DATALOG dlog2 = DATALOG_DEFAULTS;
```

Invoking the compute function:

```
dlog1.update(&dlog1);
dlog2.update(&dlog2);
```

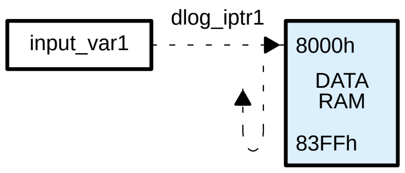
Example:

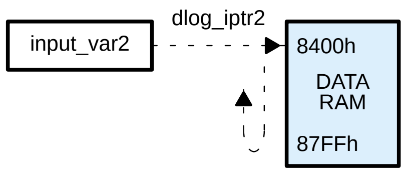
Lets instance two DATALOG objects, otherwise identical, and run four data logging variables. The following example is the c source code for the system file.

```
DATALOG dlog1= DATALOG_DEFAULTS;      /* instance the first object */
DATALOG dlog2 = DATALOG_DEFAULTS;     /* instance the second object */

main()
{

    dlog1.init(&dlog1);               /* Initialize the data_log function for dlog1 */
    dlog2.init(&dlog2);               /* Initialize the data_log function for dlog2 */

/* Since dlog1 already occupied the data buffer addressed (by default) from 0x8000 to
0x87FF, the starting buffer address for dlog2 need to set to other empty space of memory */

    dlog2.dl_buffer1_adr = 0x08800;   /* Set new starting buffer address of dlog2 */
    dlog2.dl_buffer2_adr = 0x08C00;   /* Set new starting buffer address of dlog2 */

    dlog1.dlog_iptr1 = &input1;       /* Pass inputs to dlog1 */
    dlog1.dlog_iptr2 = &input2;       /* Pass inputs to dlog1 */

    dlog2.dlog_iptr1 = &input3;       /* Pass inputs to dlog2 */
    dlog2.dlog_iptr2 = &input4;       /* Pass inputs to dlog2 */

}

void interrupt periodic_interrupt_isr()
{

    dlog1.update(&dlog1);             /* Call update function for dlog1 */
    dlog2.update(&dlog2);             /* Call update function for dlog2 */

/* This module does not have any user configurable s/w outputs and, therefore, does not
need any output parameter passing.  */

}
```

Background Information

This s/w module stores 400 realtime values of each of the selected input variables in the data RAM as illustrated in the following figures. The starting addresses of two RAM sections, where the data values are stored, are set to 8000h and 8400h.

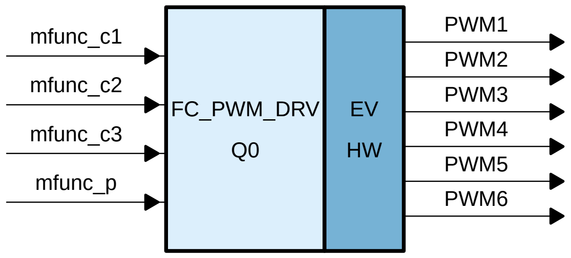




FC_PWM_DRV *Full Compare PWM Driver*

Description

This module uses the duty ratio information and calculates the compare values for generating PWM outputs. The compare values are used in the full compare unit in 24x/24xx event manager(EV). This also allows PWM period modulation.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: pwm_drv.asm

C-Callable Version File Names: F243PWM1.C, F243PWM2.ASM, F243PWM.H, F2407PWM1.C, F2407PWM2.C, F2407PWM3.ASM, F2407PWM4.ASM, F2407PWM.H, PWM.H

Item	ASM Only	C-Callable ASM	Comments
Code size	52 words	88 words † ‡ §	
Data RAM	6 words	0 words §	
Multiple instances	No	Yes	

† Multiple instances must point to distinct interfaces on the target device. Multiple instances pointing to the same PWM interface in hardware may produce undefined results. So the number of interfaces on the F241/3 is limited to one, while there can be upto two such interfaces on the LF2407.

‡ If, on the 2407, there are two interfaces concurrently linked in, then the code size will be 176 words + .cinit space + data memory space.

§ Each pre-initialized PWMGEN structure instance consumes 6 words in data memory and 8 words in the .cinit section.

Direct ASM Interface

Table 5. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	mfunc_cx (x=1,2,3)	Duty ratios for full compare unit 1, 2 and 3	Q15	8000–7FFF
	mfunc_p	PWM period modulation function	Q15	8000–7FFF
Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A
	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent

Variable Declaration:
In the system file include the following statements:

```
.ref   FC_PWM_DRV, FC_PWM _DRV _INIT           ;function call
.ref   mfunc_c1, mfunc_c2, mfunc_c3, mfunc_p   ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘pwm_drv’

Example:

```
ldp  #mfunc_c1             ;Set DP for module inputs
bldd #input_var1, mfunc_c1 ;Pass input variables
                           ;to module inputs
bldd #input_var2, mfunc_c2
bldd #input_var3, mfunc_c3
bldd #input_var4, mfunc_p
CALL FC_PWM_DRV
```

Note:

Since this is an output driver module it does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing. This s/w module calculates the compare values, which are used in the full compare unit internal to 24x/24xx device. From the compare values the device generates the PWM outputs.

C/C-Callable ASM Interface

Object Definition

The structure of the PWMGEN Interface Object is defined by the following structure definition

```
typedef struct {
int period_max;    /* PWM Period in CPU clock cycles.  Q0–Input  */
         int mfunc_p;       /* Period scaler. Q15 – Input                 */
         int mfunc_c1;      /* PWM 1&2 Duty cycle ratio. Q15, Input       */
         int mfunc_c2;      /* PWM 3&4 Duty cycle ratio. Q15, Input       */
         int mfunc_c3;      /* PWM 5&6 Duty cycle ratio. Q15, Input       */
         int (*init)();     /* Pointer to the init function               */
         int (*update)();   /* Pointer to the update function             */
         } PWMGEN ;
```

Table 6. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	mfunc_cx (x=1,2,3)	Duty ratios for full compare unit 1, 2 and 3	Q15	8000–7FFF
	mfunc_p	PWM period modulation function	Q15	8000–7FFF
Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A
	period_max	PWM period setting. Modify this constant for different PWM frequency.	Q0	Application dependent

Special Constants and Datatypes

PWMGEN
The module definition itself is created as a data type. This makes it convenient to instance an interface to the PWM Generator module.

PWMGEN _DEFAULTS
Initializer for the PWMGEN Object. This provides the initial values to the terminal variables as well as method pointers.

PWMGEN_handle
Typedef'ed to PWMGEN *

F243_FC_PWM_GEN
Constant initializer for the F243 PWM Interface.

F2407_EV1_FC_PWM_GEN
Constant initializer for the F2407 PWM Interface, EV1.

F2407_EV2_FC_PWM_GEN
Constant initializer for the F2407 PWM Interface, EV2.

Methods

void init (PWMGEN *)
Initializes the PWM Gen unit hardware.

void update(PWMGEN *)
Updates the PWM Generation hardware with the data from the PWM Structure.

Module Usage

Instantiation:
The interface to the PWM Generation Unit is instanced thus:

```
PWMGEN  gen;
```

Initialization:
To instance a pre-initialized object

```
PWMGEN  gen =PWMGEN_DEFAULTS
```

Hardware Initialization:

```
gen.init(&gen);
```

Invoking the update function:

```
gen.update(&gen);
```

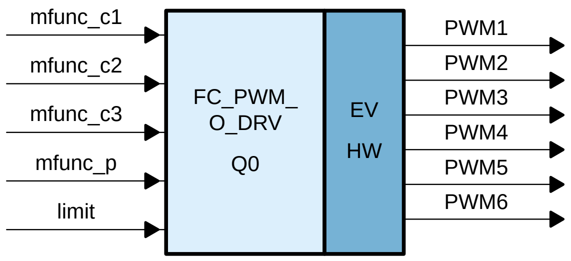
Example:
Lets instance one PWMGEN object and one SVGENMF object, (For details on SVGENMF see the SVGEN_MF.DOC.). The outputs of SVGENMF are output via the PWMGEN.

```
SVGENMF svgen= SVGEN_DEFAULTS;  /*Instance the space vector gen object */
PWMGEN  gen  = F243_FC_PWM_GEN; /*Instance the PWM interface object    */

main()
{
svgen.freq=1200;         /* Set properties for svgen */
gen.period_max=500;      /*Sets the prd reg for the Timer to 500 cycles*/
gen.init(&gen);          /* Call the hardware initialization function  */

}
void interrupt periodic_interrupt_isr()
{
sv1.calc(&sv1);          /* Call compute function for sv1 */

/* Lets output sv1.va,sv1.vb, and sv1.vc */

gen.mfunc_c1= svgen.va;  /*Connect the output of svgen to gen inputs*/
gen.mfunc_c2= svgen.vb;
gen.mfunc_c3= svgen.vc;

gen.update(&gen);        /* Call the hardware update function */
}
```

FC_PWM_O_DRV — *Full-Compare PWM Driver with Over-modulation*

Description

The module implements over-modulation technique to increase DC bus voltage utilization for a voltage source inverter. The input *limit* sets the extent of over-modulation. For example, limit = 0 means no over-modulation and limit = (timer period)/2 means maximum over-modulation.



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: pwmodrv.asm

Item	ASM Only	Comments
Code size	133 words	
Data RAM	11 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 7. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	limit	Defines the level of over modulation. This is related to the PWM timer period.	Q0	0–timer_ period/2
	Mfunc_c1	Duty ratio for PWM1/PWM2	Q15	08000h–7FFFh
	Mfunc_c2	Duty ratio for PWM3/PWM4	Q15	08000h–7FFFh
	Mfunc_c3	Duty ratio for PWM5/PWM6	Q15	08000h–7FFFh
	mfunc_p	PWM period modulation function	Q15	08000h–7FFFh
H/W Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	limit	Initial *limit* is set to 0 so that the system starts without any over-modulation. Specify *limit* for overmodulation.	Q0	0 – T1PER/2
	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent
	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A

Variable Declaration:
In the system file include the following statements:

```
.ref   FC_PWM_O_DRV
.ref   FC_PWM_O_DRV_INIT                       ;function call
.ref   Mfunc_c1, Mfunc_c2, Mfunc_c3, Mfunc_p   ;Inputs
```

Memory map:
All variables are mapped to an uninitialized named section “pwmodrv”

Example:

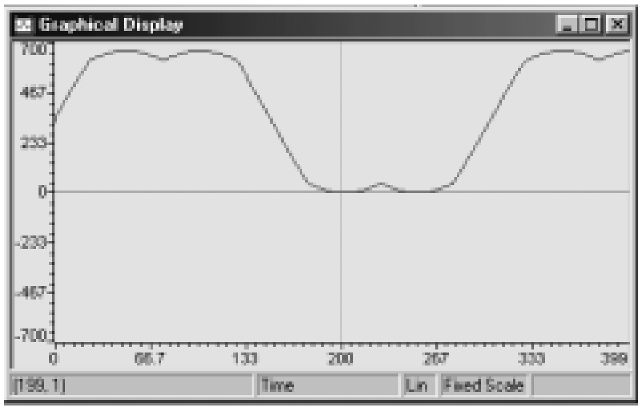
```
ldp  #mfunc_c1               ;Set DP for module inputs
bldd #input_var1, mfunc_c1   ;Pass input variables to module inputs
bldd #input_var2, mfunc_c2
bldd #input_var3, mfunc_c3
bldd #input_var4, mfunc_p

CALL FC_PWM_O_DRV
```

Background Information

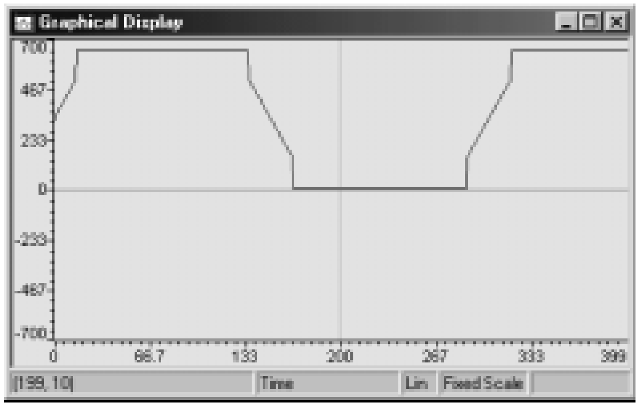
For high performance motor drive systems, full utilization of the dc bus voltage is an important factor to achieve maximum output torque under any operating conditions, and to extend the field weakening range of the motor. However, for a pulse-width modulated voltage source inverter (PWM–VSI), the maximum voltage is 78% of the six-step waveform value. Therefore, in general, a standard motor supplied from an inverter can not utilize the full DC bus voltage capability. To obtain higher DC bus voltage utilization, operating the inverter in over-modulation region is required.

This software module implements a simple but effective over-modulation scheme for PWM inverters. This module can be applied both for three phase drive (using Space Vector PWM or regular Sine PWM strategies) as well as single phase drive.

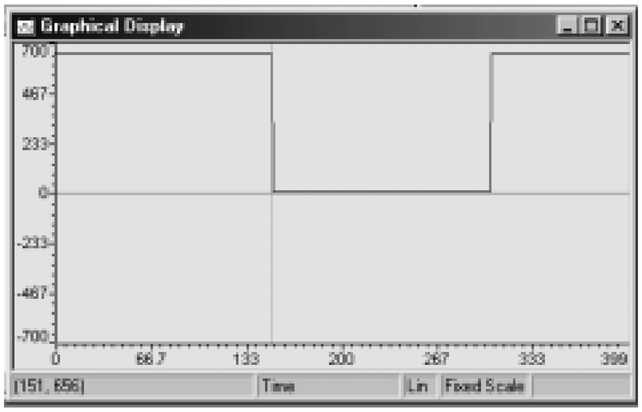
The level of over-modulation is controller by a variable called “limit”. Whenever, the ouptut waveform is within “limit”, the Compare values for PWM channels are saturated to the maximum value during the positive half of the waveform and to the minimum value during the negative half of the waveform. Figure 1 shows the effect of various values of “limit”.



(a)



(b)

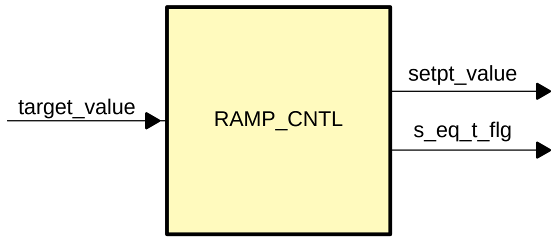


(c)

Figure 1. Implementation of Over-modulation Using the Software Module
(a) No over-modulation,
(b) Over-modulation with limit = T1PER/4,
(c) Maximum over-modulation (square wave) with limit = T1PER/2

Description

This module implements a ramp up and ramp down function. The output flag variable s_eq_t_flg is set to 7FFFh when the output variable setpt_value equals the input variable target_value.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: rmp_cntl.asm

C-Callable ASM File Names: rmp_cntl.asm, rmp_cntl.h

Item	ASM Only	C-Callable ASM	Comments
Code size	47 words	72 words†	
Data RAM	7 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized RMPCNTL structure instance consumes 8 words in the data memory and 10 words in the .cinit section.

Direct ASM Interface

Table 8. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	target_value	Desired value of the ramp	Q0	rmp_lo_limit – rmp_hi_limit
Outputs	setpt_value	Ramp output value	Q0	rmp_lo_limit – rmp_hi_limit
	s_eq_t_flg	Ramp output status flag	Q0	0 or 7FFF
Init / Config	rmp_dly_max†	Ramp step delay in number of sampling cycles	Q0	0–7FFF
	rmp_hi_limit†	Maximum value of ramp	Q0	0–7FFF
	rmp_lo_limit†	Minimum value of ramp	Q0	0–7FFF

† From the system file, initialize these variables as required by the application. From the Real-Time Code Composer window, specify *target_value* to vary the output signal *setpt_value*.

Variable Declaration:
In the system file include the following statements:

```
.ref   RAMP_CNTL, RAMP_CNTL_INIT      ; function call
.ref   target_value                   ; Inputs
.ref   rmp_dly_max, rmp_lo_limit      ; Input Parameters
.ref   rmp_hi_limit                   ; Input Parameter
.ref   setpt_value, s_eq_t_flg        ; Outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘rmp_cntl’

Example:

```
ldp  #target_value               ;Set DP for module input
bldd #input_var1, target_value ;Pass input variable to module input

CALL RAMP_CNTL

ldp    #output_var1            ;Set DP for output variable
bldd #setpt_value, output_var1 ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the RMPCNTL object is defined in the header file, rmp_cntl.h, as seen in the following:

```
typedef struct  { int  target_value;    /* Input: Target input (Q15) */
                  int  rmp_dly_max;     /* Parameter: Maximum delay rate */
                  int  rmp_lo_limit;    /* Parameter: Minimum limit (Q15) */
                  int  rmp_hi_limit;    /* Parameter: Maximum limit (Q15) */
                  int  rmp_delay_cntl;  /* Variable: Incremental delay  */
                  int  setpt_value;     /* Output: Target output (Q15) */
                  int  s_eq_t_flg;      /* Output: Flag output */
                  int  (*calc)();       /* Pointer to calculation function */
                } RMPCNTL;
```

Special Constants and Datatypes

RMPCNTL
The module definition itself is created as a data type. This makes it convenient to instance a RMPCNTL object. To create multiple instances of the module simply declare variables of type RMPCNTL.

RMPCNTL_DEFAULTS
Initializer for the RMPCNTL object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, rmp_cntl.h.

Methods

void calc(RMPCNTL *);
This default definition of the object implements just one method – the runtime compute function for ramp control. This is implemented by means of a function pointer, and the default initializer sets this to rmp_cntl_calc function. The argument to these functions is the address of the RMPCNTL object. Again, this statement is written in the header file, rmp_cntl.h.

Module Usage

Instantiation:
The following example instances two such objects

```
RMPCNTL  rmpc1, rmpc2;
```

Initialization:
To instance a pre-initialized object:

```
RMPCNTL rmpc1 = RMPCNTL_DEFAULTS;
RMPCNTL rmpc2 = RMPCNTL_DEFAULTS;
```

Invoking the compute function:

```
rmpc1.calc(&rmpc1);
rmpc2.calc(&rmpc2);
```

Example:

Lets instance two RMPCNTL objects, otherwise identical, and run two ramp controlling variables. The following example is the c source code for the system file.

```
RMPCNTL rmpc1= RMPCNTL_DEFAULTS;    /* instance the first object */
RMPCNTL rmpc2 = RMPCNTL_DEFAULTS;   /* instance the second object */

main()
{

    rmpc1.target_value = input1;        /* Pass inputs to rmpc1 */
    rmpc2.target_value = input2;        /* Pass inputs to rmpc2 */

}

void interrupt periodic_interrupt_isr()
{

    rmpc1.calc(&rmpc1);        /* Call compute function for rmpc1 */
    rmpc2.calc(&rmpc2);        /* Call compute function for rmpc2 */

output1 = rmpc1.setpt_value;   /* Access the outputs of rmpc1 */
output2 = rmpc2.setpt_value;   /* Access the outputs of rmpc2 */

}
```

Background Information

Implements the following equations:

Case 1: When *target_value* > *setpt_value*

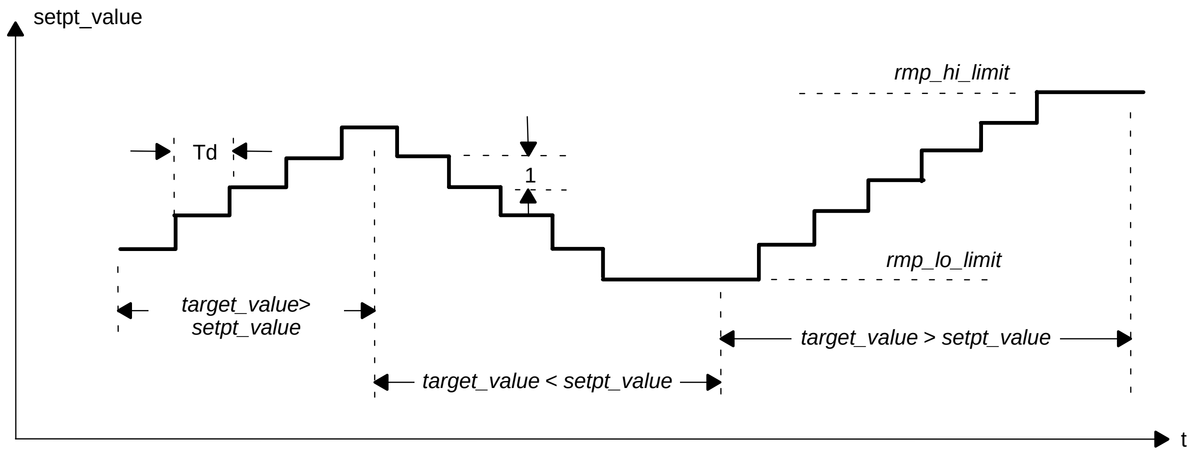
$$setpt\_value = \begin{cases} setpt\_value + 1, & \text{for } t = n \cdot Td,\ n = 1, 2, 3\ldots \\ & \text{and } (setpt\_value + 1) < rmp\_hi\_limit \\ rmp\_hi\_limit, & \text{for } (setpt\_value + 1) > rmp\_hi\_limit \end{cases}$$

where,
Td = *rmp_dly_max* **.** Ts
Ts = Sampling time period

Case 2: When *target_value* < *setpt_value*

$$setpt\_value = \begin{cases} setpt\_value - 1, & \text{for } t = n \cdot Td,\ n = 1, 2, 3\ldots.. \\ & \text{and } (setpt\_value - 1) > rmp\_lo\_limit \\ rmp\_lo\_limit, & \text{for } (setpt\_value - 1) < rmp\_lo\_limit \end{cases}$$

where,
Td = *rmp_dly_max* **.** Ts
Ts = Sampling time period


setpt_value
rmp_hi_limit
Td
1
rmp_lo_limit
target_value> setpt_value
target_value > setpt_value
target_value < setpt_value
t

Example:

setpt_value = 0 (initial value), target_value = 1000 (user specified),
rmp_dly_max = 500 (user specified), sampling loop time period Ts = 0.000025 Sec.

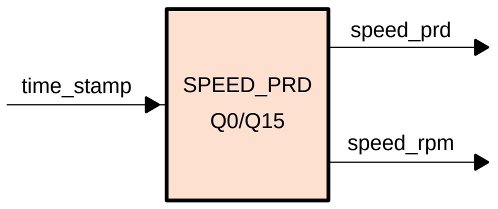
This means that the time delay for each ramp step is Td = 500x0.000025 = 0.0125 Sec.
Therefore, the total ramp time will be Tramp = 1000x0.0125 Sec = 12.5 Sec

SPEED_PRD

Speed Calculator Based on Period Measurement

Description

This module calculates the motor speed based on a signal's period measurement. Such a signal, for which the period is measured, can be the periodic output pulses from a motor speed sensor.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: speed_pr.asm

C-Callable Version File Names: speed_pr.asm, speed_pr.h

Item	ASM Only	C-Callable ASM	Comments
Code size	55 words	64 words†	
Data RAM	13 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† Each pre-initialized SPEED_MEAS structure instance consumes 9 words in the data memory and 11 words in the .cinit section.

Direct ASM Interface

Table 9. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	time_stamp	Captured base timer counter value corresponding to the periodic edges of the sensed signal.	Q0	0–FFFF
Outputs	speed_prd	Normalized motor speed	Q15	0–7FFF
	speed_rpm	Motor speed in revolution per minute	Q0	0–rpm_max
Init / Config	rpm_max	Speed of normalization. The value chosen should be equal to or greater than the maximum motor speed.	Q0	Specified by user
	speed_scaler	Scaling constant. Use the Excel file to calculate this.	Q0	System dependent
	shift	Number of left shift less 1 required for max accuracy of 32bit/16bit division used for speed calculation. Use the Excel file to calculate this. When speed_scaler is calculated as 1, shift will be –1. In that case do not apply any left shift on the result of the 32bit/16bit division.	Q0	System dependent

Variable Declaration: In the system file include the following statements:

```
.ref   SPEED_PRD, SPEED_PRD _INIT                    ;function call
.ref    time_stamp, rpm_max, speed_scaler, shift    ;input
.ref    speed_prd, speed_rpm                        ;output
```

Memory map: All variables are mapped to an uninitialized named section ‘speedprd’

Example:

```
ldp  # time_stamp              ;Set DP for module input
bldd #input_var1, time_stamp   ;Pass input to module input
CALL SPEED_PRD
ldp  #output_var1              ;Set DP for output variables
bldd #speed_prd, output_var1   ;Pass module outputs to output
                               ;variables
bldd #speed_rpm, output_var2
```

C/C-Callable ASM Interface

Object Definition

The structure of the SPEED_MEAS object is defined by the following structure definition

```
typedef struct {
  int time_stamp_new;   /*Variable: New ‘Timestamp’ corresponding to a capture event*/
  int time_stamp_old;   /*Variable: Old ‘Timestamp’ corresponding to a capture event*/
  int time_stamp;         /*Input: Current ‘Timestamp’ corresponding to a capture event*/
  int shift;                /*Parameter: For maximum accuracy of 32bit/16bit division*/
  int speed_scaler;       /*Parameter: Scaler converting 1/N cycles to a Q15 speed*/
  int speed_prd;          /*Output: speed in per–unit
  int rpm_max;            /*Parameter: Scaler converting Q15 speed to rpm (Q0) speed*/
  int speed_rpm;        /*Output: speed in r.p.m.
  int (*calc) ();         /*Pointer to the calculation function*/
  } SPEED_MEAS;           /*Data type created*/
```

Table 10. Module Terminal Variables

	Name	Description	Format	Range
Inputs	time_stamp	Captured base timer counter value corresponding to the periodic edges of the sensed signal.	Q0	0–FFFF
Outputs	speed_prd	Normalized motor speed	Q15	0–7FFF
	speed_rpm	Motor speed in revolution per minute	Q0	0–rpm_max
Init / Config	rpm_max	Speed of normalization. The value chosen should be equal to or greater than the maximum motor speed.	Q0	Specified by user
	speed_scaler	Scaling constant. Use the Excel file to calculate this.	Q0	System dependent
	shift	Number of left shift less 1 required for max accuracy of 32bit/16bit division used for speed calculation. Use the Excel file to calculate this. When speed_scaler is calculated as 1, shift will be –1. In that case, do not apply any left shift on the result of the 32 bit/16 bit division.	Q0	System dependent

Special Constants and Datatypes

SPEED_MEAS

The module definition itself is created as a data type. This makes it convenient to instance a Space Vector Generation module. To create multiple instances of the module simply declare variables of type SVGENMF.

SPEED_PR_MEAS_DEFAULTS
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(SPEED_MEAS *)
Pointer to the speed calculation function.

Module Usage

Instantiation:

```
  SPEED_MEAS  shaftSpeed;
```

Initialization:
To instance a pre-initialized object

```
  SPEED_MEAS  shaftSpeed=SPEED_PR_MEAS_DEFAULTS;
```

Invoking the compute function:

```
shaftSpeed.calc(&shaftSpeed);
```

Example:

```
/*--------------------------------------------------------------------
Pre initialized declaration for the speed measurement object.
--------------------------------------------------------------------*/
        SPEED_MEAS shaftSpeed=SPEED_PR_MEAS_DEFAULTS;

/*--------------------------------------------------------------------
Declaration for the capture driver. For more details see the CAP_DRV
document.
--------------------------------------------------------------------*/
        CAPTURE cap=CAPTURE_DEFAULTS;
main()
{

/*--------------------------------------------------------------------
    Initialize the capture interface
--------------------------------------------------------------------*/
        cap.init(&cap);

}

void periodic_interrupt_isr()
{
/*-------------------------------------------------------------------
Call the capture driver read function. Note, that this func returns
the status, as the return value, NOT the time_stamp. The time_stamp
is returned directly into the CAPTURE object structure.
--------------------------------------------------------------------*/
  if((cap.read(&cap))==0)  /* Call the capture read function */
  {
 shaftSpeed.time_stamp=cap.time_stamp; /* Read out new time stamp */
 shaftSpeed.calc(&shaftSpeed);        /* Call the speed calulator */
  }

}
```

Background Information

A low cost shaft sprocket with n teeth and a Hall effect gear tooth sensor is used to measure the motor speed. Figure 2 shows the physical details associated with the sprocket. The Hall effect sensor outputs a square wave pulse every time a tooth rotates within its proximity. The resultant pulse rate is n pulses per revolution. The Hall effect sensor output is fed directly to the 24x/24xx Capture input pin. The capture unit will capture (the value of it’s base timer counter) on either the rising or the falling edges(whichever is specified) of the Hall effect sensor output. The captured value is passed to this s/w module through the variable called *time_stamp*.

In this module, every time a new input *time_stamp* becomes available it is compared with the previous *time_stamp*. Thus, the tooth-to-tooth period (t_2–t_1) value is calculated. In order to reduce jitter or period fluctuation, an average of the most recent n period measurements can be performed each time a new pulse is detected.

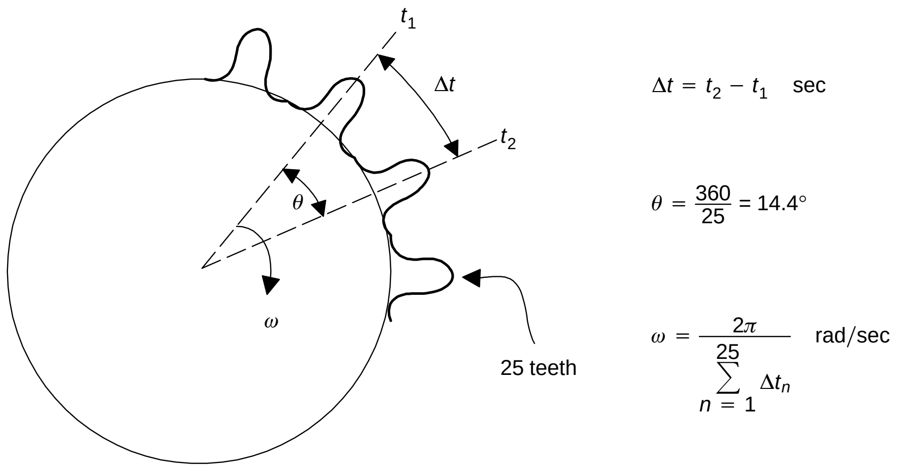

t_1
Δt
t_2
θ
ω
25 teeth
$\Delta t = t_2 - t_1$ sec
$\theta = \frac{360}{25} = 14.4°$
$\omega = \frac{2\pi}{\sum_{n=1}^{25} \Delta t_n}$ rad/sec

Figure 2. Speed Measurement With a Sprocket

From the two consecutive *time_stamp* values the difference between the captured values are calculated as,

Δ = time_stamp(new) – time_stamp(old)

Then the time period in sec is given by,

$$\Delta t = t_2 - t_1 = K_p \times T_{CLK} \times \Delta$$

where,

K_P = Prescaler value for the Capture unit time base

T_{CLK} = CPU clock period in sec

From Figure 2, the angle θ in radian is given by,

$$\theta = \frac{2\pi}{n}$$

where,

n = number of teeth in the sprocket, i.e., the number of pulses per revolution

Then the speed ω in radian/sec and the normalized speed ω_N are calculated as,

$$\omega = \frac{\theta}{\Delta t} = \frac{2\pi}{n\Delta t} = \frac{2\pi}{n \times K_p \times T_{CLK} \times \Delta}$$

$$\Rightarrow \omega_N = \frac{\omega}{\omega_{\max}} = \frac{\omega}{2\pi\left(\frac{1}{n \times K_P \times T_{CLK}}\right)} = \frac{1}{\Delta}$$

Where, ω_{max} is the maximum value of ω which occurs when Δ=1. Therefore,

$$\omega_{\max} = \frac{2\pi}{nK_P T_{CLK}}$$

For, n=25, K_P=32 and T_{CLK}=50x10^{-9} sec (20MHz CPU clock), the normalized speed ω_N is given by,

$$\omega_N = \frac{\omega}{2\pi(25000)} = \frac{1}{\Delta}$$

The system parameters chosen above allows maximum speed measurement of 1500,000 rpm. Now, in any practical implementation the maximum motor speed will be significantly lower than this maximum measurable speed. So, for example, if the motor used has a maximum operating speed of 23000 rpm, then the calculated speed can be expressed as a normalized value with a base value of normalization of at least 23000 RPM. If we choose this base value of normalization as 23438 rpm, then the corresponding base value of normalization, in rad/sec, is,

$$\omega_{max1} = \frac{23438 \times 2\pi}{60} \approx 2\pi(390)$$

Therefore, the scaled normalized speed is calculated as,

$$\omega_{N1} = \frac{\omega}{2\pi(390)} \approx \frac{64}{\Delta} = 64 \times \omega_N = speed\_scaler \times \omega_N$$

This shows that in this case the scaling factor is 64.

The speed, in rpm, is calculated as,

$$N_1 = 23438 \times \omega_{N1} = 23438 \times \frac{64}{\Delta} = rpm\_max \times \omega_{N1}$$

The capture unit in 24x/24xx allows accurate time measurement (in multiples of clock cycles and defined by a prescaler selection) between events. In this case the events are selected to be the rising edge of the incoming pulse train. What we are interested in is the delta time between events and hence for this implementation Timer 1 is allowed to free run with a prescale of 32 (1.6uS resolution for 20MHz CPU clock) and the delta time Δ, in scaled clock counts, is calculated as shown in Figure 3.

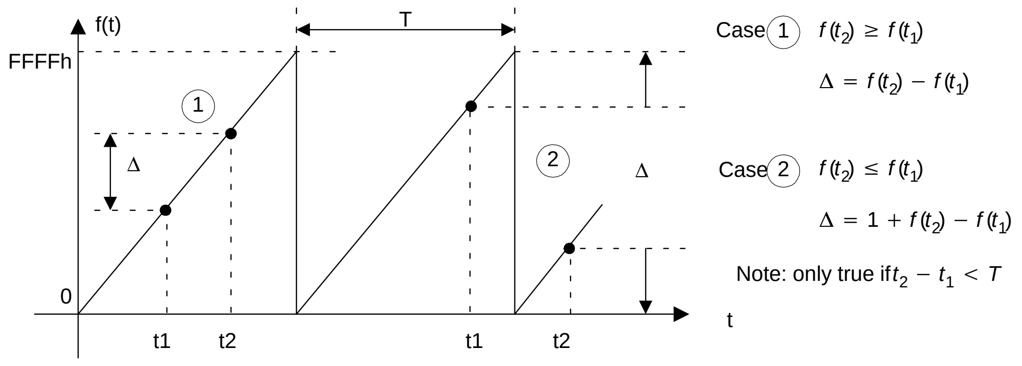

f(t)
T
FFFFh
1
Δ
2
Δ
0
t1
t2
t1
t2
t
Case 1 $f(t_2) \geq f(t_1)$
$\Delta = f(t_2) - f(t_1)$
Case 2 $f(t_2) \leq f(t_1)$
$\Delta = 1 + f(t_2) - f(t_1)$
Note: only true if $t_2 - t_1 < T$

Figure 3. Calculation of Speed

In Figure 3, the vertical axis *f(t)* represents the value of the Timer counter which is running in continuous up count mode and resetting when the period register = FFFFh. Note that two cases need to be accounted for: the simple case where the Timer has not wrapped around and where it has wrapped around. By keeping the current and previous capture values it is easy to test for each of these cases.

Once a "robust" period measurement is extracted from the averaging algorithm, the speed is calculated using the appropriate equations explained before. In order to maintain high precision in the calculation for the full range of motor speeds, a 32-bit/16-bit division is performed as shown in Figure 4 in the following.

$$\frac{1}{period} = \frac{7FFFFFFF(Q31)}{period(Q0)} = speed(Q31 \rightarrow 32bit)$$

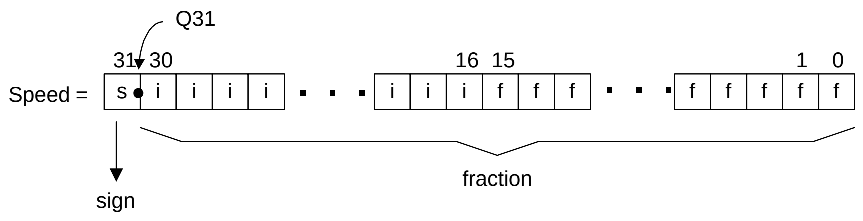


Figure 4. 32-Bit/16-Bit Division

Once complete the result is a 32-bit value in $Q31$ format. This value is subsequently scaled to a 16 bit, $Q15$ format value for later calculation of the speed error (see Figure 4).

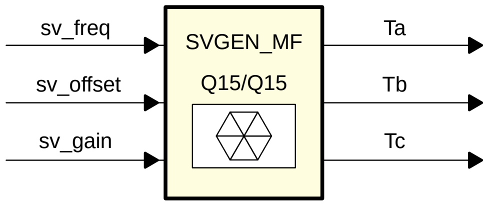
Table 11. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Δ	event_period
ω_N	speed_prd_max
ω_{N1}	speed_prd
N_1	speed_rpm

SVGEN_MF *Space Vector Generator (Magnitude/Frequency Method)*

Description

This module calculates the appropriate duty ratios needed to generate a given stator reference voltage using space vector PWM technique. The stator reference voltage is described by it’s magnitude and frequency.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: svgen_mf.asm

C-Callable Version File Name: svgen_mf.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	427 words	454 words†	
Data RAM	16 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized SVGENMF structure consumes 11 words in the .cinit section instance and 9 words in data memory.

Direct ASM Interface

Table 12. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	sv_freq	Normalized frequency of reference voltage vector.	Q15	8000–7FFF
	sv_gain	Normalized gain of the reference voltage vector.	Q15	8000–7FFF
	sv_offset	Normalized offset in the reference voltage vector	Q15	8000–7FFF
Outputs	Ta	Duty ratio of PWM1(CMPR1 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF
	Tb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF
	Tc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   SVGEN_MF, SVGEN_MF _INIT                   ;function call

.ref   sv_freq, sv_gain, sv_offset, Ta, Tb, Tc    ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘svgen_mf’

Example:

```
ldp  #sv_freq              ;Set DP for module input
bldd #input_var1, sv_freq  ;Pass input variables to module inputs
bldd #input_var2, sv_gain
bldd #input_var2, sv_offset

CALL SVGEN_MF

ldp  #output_var1          ;Set DP for output variable
bldd #Ta, output_var1      ;Pass module outputs to output variables
bldd #Tb, output_var2
bldd #Tc, output_var3
```

C/C-Callable ASM Interface

Object Definition

The structure of the SVGENMF object is defined by the following structure definition

```
/*---------------------------------------------------------------------------
Define the structure of the SVGENMF
(Magnitude and angular velocity based Space Vector Waveform Generator)
-----------------------------------------------------------------------------*/


typedef struct  { int gain;      /* Waveform amplitude Q15 Input         */
                  int freq;      /* Frequency setting  Q15 Input         */
                  int freq_max;  /* Frequency setting  Q0  Input         */
                  int alpha;     /* Internal var – Sector angle history  */
                  int sector;    /* Internal var – Sector number history */
                  int va;        /* Phase A output Q15                   */
                  int vb;        /* Phase B output Q15                   */
                  int vc;        /* Phase C output Q15                   */
                  int (*calc)(); /* Pointer to calculation function      */


                } SVGENMF;
```

Table 13. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	freq	Fraction of Frequency of reference voltage vector.	Q15	8000–7FFF
	freq_max	Frequency of reference voltage vector.	Q0	8000–7FFF
	gain	Required gain for the desired reference voltage vector.	Q15	8000–7FFF
Outputs	va	Duty ratio of PWM1(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF
	vb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF
	vc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000–7FFF

Special Constants and Datatypes

SVGENMF
The module definition itself is created as a data type. This makes it convenient to instance a Space Vector Generation module. To create multiple instances of the module simply declare variables of type SVGENMF.

SVGENDQ_handle
Typedef'ed to SVGENMF *

SVGENMF_DEFAULTS
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(SVGENMF_handle)
The default definition of the object implements just one method – the runtime compute function for the generation of the space vector modulation functions. This is implemented by means of a function pointer, and the default initializer sets this to svgenmf_calc. The argument to this function is the address of the SVGENMF object.

Module Usage

Instantiation:
The following example instances two such objects:

```
SVGENMF   sv1,sv2;
```

Initialization:
To instance a pre-initialized object

```
SVGENMF  sv1=SVGEN_DEFAULTS,sv2=SVGEN_DEFAULTS;
```

Invoking the compute function:

```
sv1.calc(&sv1);
```

Example:
Lets instance two SVGENMF objects, otherwise identical, but running with different freq values.

```
SVGENMF sv1=SVGEN_DEFAULTS; /* Instance the first object */
SVGENMF sv2=SVGEN_DEFAULTS; /* Instance the second object*/

main()
{
    sv1.freq=1200;  /* Set properties for sv1 */
    sv2.freq=1800;  /* Set properties for sv2 */
}
void interrupt periodic_interrupt_isr()
{
    sv1.calc(&sv1); /* Call compute function for sv1 */
    sv2.calc(&sv2); /* Call compute function for sv2 */

    x=sv1.va;       /* Access the outputs of sv1 */
    y=sv1.vb;
    z=sv1.vc;

    p=sv2.va;       /* Access the outputs of sv2 */
    q=sv2.vb;
    r=sv2.vc;

/* Do something with the outputs */

    }
```

Background Information

The Space Vector Pulse Width Modulation (SVPWM) refers to a special switching sequence of the upper three power devices of a three-phase voltage source inverters (VSI) used in application such as AC induction and permanent magnet synchronous motor drives. This special switching scheme for the power devices results in 3 pseudo-sinusoidal currents in the stator phases.

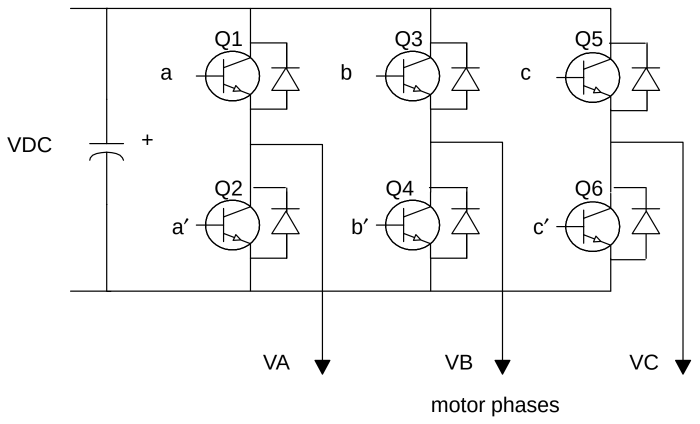


Figure 5. Power Circuit Topology for a Three-Phase VSI

It has been shown that SVPWM generates less harmonic distortion in the output voltages or currents in the windings of the motor load and provides more efficient use of DC supply voltage, in comparison to direct sinusoidal modulation technique.

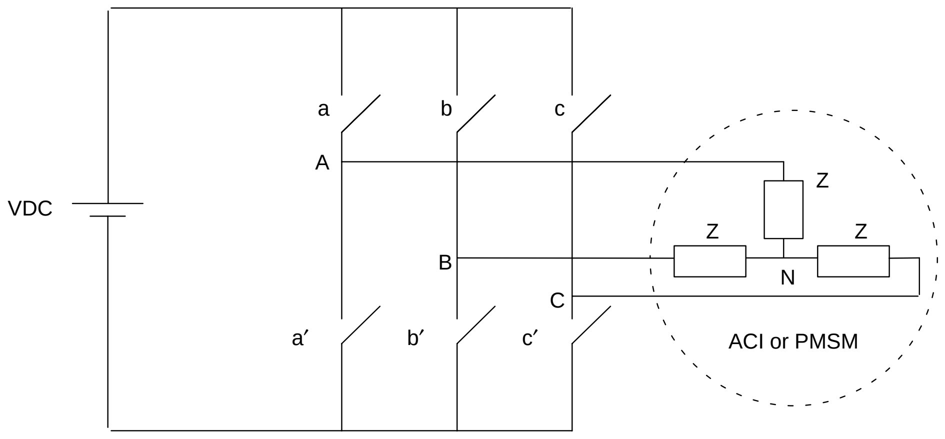


Figure 6. Power Bridge for a Three-Phase VSI

For the three phase power inverter configurations shown in Figure 5 and Figure 6, there are eight possible combinations of on and off states of the upper power transistors. These combinations and the resulting instantaneous output line-to-line and phase voltages, for a dc bus voltage of $V_{DC,}$ are shown in Table 14.

Table 14. Device On/Off Patterns and Resulting Instantaneous Voltages of a 3-Phase Power Inverter

c	b	a	V_{AN}	V_{BN}	V_{CN}	V_{AB}	V_{BC}	V_{CA}
0	0	0	0	0	0	0	0	0
0	0	1	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}/3$	V_{DC}	0	$-V_{DC}$
0	1	0	$-V_{DC}/3$	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}$	V_{DC}	0
0	1	1	$V_{DC}/3$	$V_{DC}/3$	$-2V_{DC}/3$	0	V_{DC}	$-V_{DC}$
1	0	0	$-V_{DC}/3$	$-V_{DC}/3$	$2V_{DC}/3$	0	$-V_{DC}$	V_{DC}
1	0	1	$V_{DC}/3$	$-2V_{DC}/3$	$V_{DC}/3$	V_{DC}	$-V_{DC}$	0
1	1	0	$-2V_{DC}/3$	$V_{DC}/3$	$V_{DC}/3$	$-V_{DC}$	0	V_{DC}
1	1	1	0	0	0	0	0	0

The quadrature quantities (in d–q frame) corresponding to these 3 phase voltages are given by the general Clarke transform equation:

$$V_{ds} = V_{AN}$$

$$V_{qs} = \frac{(2V_{BN} + V_{AN})}{\sqrt{3}}$$

In matrix from the above equation is also expressed as,

$$\begin{bmatrix} V_{ds} \\ V_{qs} \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} V_{AN} \\ V_{BN} \\ V_{CN} \end{bmatrix}$$

Due to the fact that only 8 combinations are possible for the power switches, V_{ds} and V_{qs} can also take only a finite number of values in the (d–q) frame according to the status of the transistor command signals (c,b,a). These values of V_{ds} and V_{qs} for the corresponding instantaneous values of the phase voltages (V_{AN}, $V_{BN,}$ V_{CN}) are listed in Table 15.

Table 15. Switching Patterns, Corresponding Space Vectors, and their (d–q) Components

c	b	a	V_{ds}	V_{qs}	Vector
0	0	0	0	0	O_0
0	0	1	$\frac{2V_{DC}}{3}$	0	U_0
0	1	0	$-\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{120}
0	1	1	$\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{60}
1	0	0	$-\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{240}
1	0	1	$\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{300}
1	1	0	$-\frac{2V_{DC}}{3}$	0	U_{180}
1	1	1	0	0	O_{111}

These values of V_{ds} and $V_{qs,}$ listed in Table 15, are called the (d–q) components of the basic space vectors corresponding to the appropriate transistor command signal (c,b,a). The space vectors corresponding to the signal (c,b,a) are listed in the last column in Table 15. For example, (c,b,a)=001 indicates that the space vector is U_0.The eight basic space vectors defined by the combination of the switches are also shown in Figure 7.

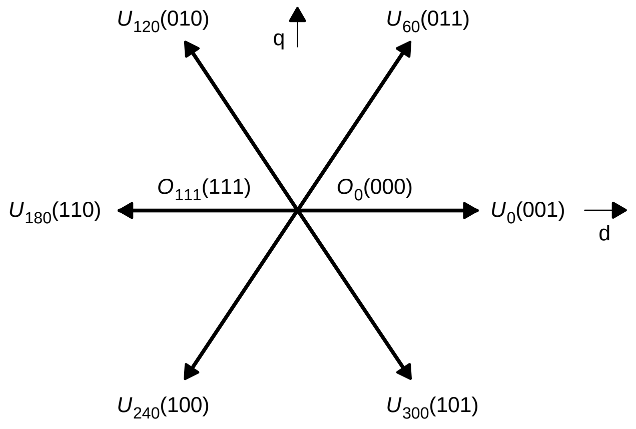


Figure 7. Basic Space Vectors

In Figure 7, vectors corresponding to states 0 (000) and 7 (111) of the switching variables are called the zero vectors.

Decomposing the reference voltage vector V*

The objective of Space Vector PWM technique is to approximate a given stator reference voltage vector V* by combination of the switching pattern corresponding to the basic space vectors. The reference voltage vector V* is obtained by mapping the desired three phase output voltages(line to neutral) in the (d–q) frame through the Clarke transform defined earlier. When the desired output phase voltages are balanced three phase sinusoidal voltages, V* becomes a vector rotating around the origin of the (d–q) plane with a frequency corresponding to that of the desired three phase voltages.

The magnitude of each basic space vector, as shown in Figure 8, is normalized by the maximum value of the phase voltages. Therefore, when the maximum bus voltage is V_{DC}, the maximum line to line voltage is also V_{DC}, and so the maximum phase voltage(line to neutral) is $V_{DC}/\sqrt{3}$. From Table 15, the magnitude of the basic space vectors is $2V_{DC}/3$. When this is normalized by the maximum phase voltage($V_{DC}/\sqrt{3}$), the magnitude of the basic space vectors becomes $2/\sqrt{3}$. These magnitudes of the basic space vectors are indicated in Figure 8.

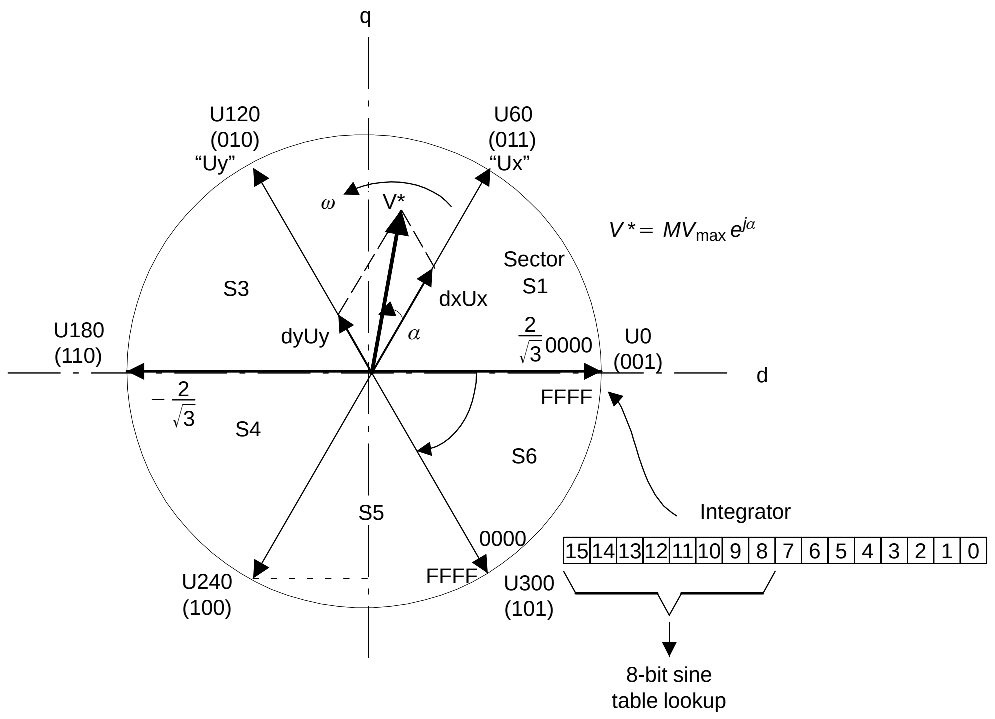


Figure 8. Projection of the Reference Voltage Vector

Representing the reference vector V* with the basic space vectors requires precise control of both the vector magnitude M (also called the modulation index) and the angle α. The aim here is to rotate V* in the d–q plane at a given angular speed (frequency) ω. The vector magnitude M controls the resultant peak phase voltage generated by the inverter.

In order to generate the reference vector V*, a time average of the associated basic space vectors is required, i.e. the desired voltage vector V* located in a given sector, can be synthesized as a linear combination of the two adjacent space vectors, Ux and Uy which frame the sector, and either one of the two zero vectors. Therefore,

$$V^* = dxUx + dyUy + dzUz$$

where Uz is the zero vector, and dx, dy and dz are the duty ratios of the states X, Y and Z within the PWM switching interval. The duty ratios must add to 100% of the PWM period, i.e: dx + dy + dz = 1.

Vector V* in Figure 8 can also be written as:

$$V^* = MV_{max}\, e^{j\alpha} = dxUx + dyUy + dzUz$$

where M is the modulation index and V_{max} is the maximum value of the desired phase voltage.

By projecting V* along the two adjacent space vectors Ux and Uy, we have,

$$\begin{cases} MV_{max}\cos\alpha = dx|Ux| + dy|Uy|\cos 60^\circ \\ MV_{max}\sin\alpha = dy|Uy|\sin 60^\circ \end{cases}$$

Since the voltages are normalized by the maximum phase voltage, V_{max}=1. Then by knowing |Ux| = |Uy| = 2/√3 (when normalized by maximum phase voltage), the duty ratios can be derived as,

$$dx = M\sin(60 - \alpha)$$

$$dy = M\sin(\alpha)$$

These same equations apply to any sector, since the d–q reference frame, which has here no specific orientation in the physical space, can be aligned with any space vector.

Implementation of sin function

In this implementation the angular speed ω is controlled by a precision frequency generation algorithm which relies on the modulo nature (i.e. wrap-around) of a finite length register, called Integrator in Figure 8. The upper 8 bits of this integrator (a data memory location in 24x/24xx) is used as a pointer to a 256 word Sine lookup table. By adding a fixed value (step size) to this register, causes the 8 bit pointer to cycle at a constant rate through the Sine table. In effect we are integrating angular velocity to give angular position. At the end limit the pointer simply wraps around and continues at the next modulo value given by the step size. The rate of cycling through the table is very easily and accurately controlled by the value of step size.

As shown in Figure 8, sine of α is needed to decompose the reference voltage vector onto the basic space vectors of the sector the voltage vector is in. Since this decomposition is identical among the six sectors, only a 60° sine lookup table is needed. In order to complete one revolution (360^{o}) the sine table must be cycled through 6 times.

For a given step size the angular frequency (in cycles/sec) of V* is given by:

$$\omega = \frac{STEP \times f_s}{6 \times 2^m}$$

where

f_S = sampling frequency (i.e. PWM frequency)

STEP = angle stepping increment

m = # bits in the integration register.

For example, if f_S = 24KHz, *m*=16 bits & STEP ranges from 0à2048 then the resulting angular frequencies will be as shown in Table 16.

Table 16. Frequency Mapping

STEP	Freq (Hz)	STEP	Freq (Hz)	STEP	Freq (Hz)
1	0.061	600	36.62	1700	103.76
20	1.22	700	42.72	1800	109.86
40	2.44	800	48.83	1900	115.97
60	3.66	900	54.93	2000	122.07
80	4.88	1000	61.04	2100	128.17
100	6.10	1100	67.14	2200	134.28

From the table it is clear that a STEP value of 1 gives a frequency of 0.061Hz, this defines the frequency setting resolution, i.e. the actual line voltage frequency delivered to the AC motor can be controlled to better than 0.1 Hz.

For a given f_S the frequency setting resolution is determined by *m* the number of bits in the integration register. Table 17 shows the theoretical resolution which results from various sizes of m.

Table 17. Resolution of Frequency Mapping

m (# bits)	Freq res (Hz)	m (# bits)	Freq res (Hz)
8	15.6250	17	0.0305
12	0.9766	18	0.0153
14	0.2441	19	0.0076
16	0.0610	20	0.0038

Another important parameter is the size of the lookup table. This directly effects the harmonic distortion produced in the resulting synthesized sine wave. As mentioned previously a 256 entry sine table is used which has a range of 60°. This gives an angle lookup resolution of 60° / 256 = 0.23°. The table entries are given in Q15 format and a summarized version is shown below.

```
;--------------------------------------------------------
;No. Samples: 256,  Angle Range: 60, Format: Q15
;--------------------------------------------------------
;            SINVAL ;   Index  Angle    Sin(Angle)
;--------------------------------------------------------
STABLE  .word   0     ;  0    0        0.00
        .word   134   ;  1    0.23     0.00
        .word   268   ;  2    0.47     0.01
        .word   402   ;  3    0.70     0.01
        .word   536   ;  4    0.94     0.02
        .word   670   ;  5    1.17     0.02
          ”         ”       ”       ”    ”
          ”         ”       ”       ”    ”
          ”         ”       ”       ”    ”
        .word   28106 ;  252  59.06    0.86
        .word   28175 ;  253  59.30    0.86
        .word   28243 ;  254  59.53    0.86
        .word   28311 ;  255  59.77    0.86
```

Realization of the PWM Switching Pattern

Once the PWM duty ratios dx, dy and dz are calculated, the appropriate compare values for the compare registers in 24x/24xx can be determined. The switching pattern in Figure 9 is adopted here and is implemented with the Full Compare Units of 24x/24xx. A set of 3 new compare values, Ta, Tb and Tc, need to be calculated every PWM period to generate this switching pattern.

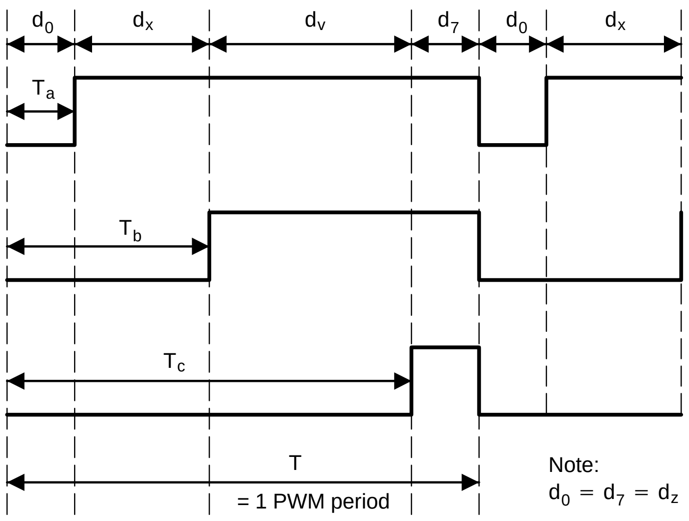


Figure 9. PWM Output Switching Pattern

From Figure 9, it can be seen:

$$Ta = \frac{(T - dx - dy)}{2}$$

$$Tb = dx + Ta$$

$$Tc = T - Ta$$

If we define an intermediate variable T1 using the following equation:

$$T1 = \frac{T - dx - dy}{2}$$

Then for different sectors Ta, Tb and Tc can be expressed in terms of T1. Table 18 depicts this determination.

Table 18. Calculation of Duty Cycle for Different Sectors

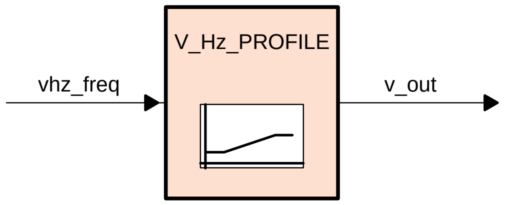
Sector	U_0, U_{60}	U_{60}, U_{120}	U_{120}, U_{180}	U_{180}, U_{240}	U_{240}, U_{300}	U_{300}, U_0
Ta	T1	dy+Tb	T–Tb	T–Tc	dx+Tc	T1
Tb	dx+Ta	T1	T1	dy+Tc	T–Tc	T–Ta
Tc	T–Ta	T–Tb	dx+Tb	T1	T1	dy+Ta

The switching pattern shown in Figure 9 is an asymmetric PWM implementation. However, 24x/24xx devices can also generate symmetric PWM. Little change to the above implementation is needed to accommodate for this change. The choice between the symmetrical and asymmetrical case depends on the other care-about in the final implementation.

V_HZ_PROFILE — *Volts/Hertz Profile for AC Induction Motor*

Description

This module generates an output command voltage for a specific input command frequency according to the specified volts/hertz profile. This is used for variable speed implementation of AC induction motor drives.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent/Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: vhz_prof.asm

C-Callable Version File Names: vhzprof.asm, vhzprof.h

Item	ASM Only	C-Callable ASM	Comments
Code size	42 words	48 words†	
Data RAM	9 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† Each pre-initialized VHZPROFILE struction consumes 10 words in the .cinit section instance and 8 words in data memory.

Direct ASM Interface

Table 19. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	vhz_freq	Command frequency of the stator voltage	Q15	0–7FFF
Outputs	v_out	Command stator output voltage	Q15	0–7FFF
Init / Config	FL†	Low frequency point on v/f profile.	Q15	Application dependent
	FH†	High frequency point on v/f profile.	Q15	Application dependent
	Fmax†	Maximum frequency	Q15	Application dependent
	vf_slope†	Slope of the v/f profile	Q12	Application dependent
	Vmax†	Voltage corresponding to FH	Q15	Application dependent
	Vmin†	Voltage corresponding to FL	Q15	Application dependent

† These parameters are initialized to some default values in the module initialization routine. Initialize these from the system file if the default values are not used.

Variable Declaration:
In the system file include the following statements:

```
.ref   V_Hz_PROFILE, V_Hz_PROFILE _INIT    ;function call

.ref   vhz_freq, v_out                     ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section 'vhz_prof'

Example:

```
ldp  #vhz_freq                ;Set DP for module input
bldd #input_var1, vhz_freq    ;Pass input variable to module
                              ;input

CALL V_Hz_PROFILE

ldp  #output_var1             ;Set DP for output variable
bldd #v_out, output_var1      ;Pass module output to output
                              ; variable
```

C/C-Callable ASM Interface

Object Definition

The object is defined as

```
typedef struct { int freq;   /* Frequency input Q15 */
                 int fl;     /* Freq below which vout=vmin:Q15 Input  */
                 int fh;     /* Freq above which vout=vmax Q15 Input  */
                 int slope;  /* Slope of the Vhz profile:  Q15 Input  */
                 int vmax;   /* Voltage output above fmax  Q15 Input  */
                 int vmin;   /* Voltage output below fmin  Q15 Input  */
                 int vout;   /* Computed output voltage    Q15 Output */
                 int (*calc)();  /* Ptr to the calculation function   */
               } VHZPROFILE;
```

Table 20. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	freq	Command frequency of the stator voltage	Q15	0–7FFF
Outputs	vout	Command stator output voltage	Q15	0–7FFF
Init / Config	fl†	Low frequency point on v/f profile.	Q15	Application dependent
	fh†	High frequency point on v/f profile.	Q15	Application dependent
	slope†	Slope of the v/f profile	Q12	Application dependent
	vmax†	Voltage corresponding to fl	Q15	Application dependent
	vmin†	Voltage corresponding to fh	Q15	Application dependent

† These parameters are initialized to some default values in the module initialization routine. Initialize these from the system file if the default values are not used.

Special Constants and Datatypes

VHZPROFILE
The module definition itself is created as a data type. This makes it convenient to instance a VHZ Profile module. To create multiple instances of the module simply declare variables of type VHZPROFILE.

DEFAULT_PROFILE
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(VHZPROFILE *)
The only method implemented for this object is the runtime compute function for the calculation of the vout value depending on the object parameters. The argument to this function is the address of the VHZPROFILE object.

Module Usage

Instantiation:
The following example instances two such objects:

```
VHZPROFILE   vhz1,vhz2;
```

Initialization:
To instance a pre-initialized object

```
VHZPROFILE   vhz1=DEFAULT_PROFILE;
```

Invoking the compute function:

```
vhz1.calc(&vhz1);
```

Example:
Lets instance two SVGENMF objects, otherwise identical, but running with different freq values. These SVGENMF objects need the computed value of the envelope for the SVGEN waveforms, and this is computed by the VHZPROFILE objects.

```
  SVGENMF sv1=SVGEN_DEFAULTS;  /* Instance the first object */
  SVGENMF sv2=SVGEN_DEFAULTS;  /* Instance the second object*/

  VHZPROFILE vhz1=DEFAULT_PROFILE;
  VHZPROFILE vhz2=DEFAULT_PROFILE;

main()
{
  sv1.freq=1200;               /* Set properties for sv1 */
  sv2.freq=1800;               /* Set properties for sv2 */
}

void interrupt periodic_interrupt_isr()
{
  vhz1.freq=sv1.freq;          /* Connect the sv1, sv2 freq to vhz1 and vhz2 */
  vhz1.freq=sv1.freq;

  vhz2.calc(&vhz1);            /* Call the compute functions */
  vhz2.calc(&vhz1);

  sv1.gain=vhz1.gain;          /* Pass the computed output voltages back to the svgens */

  sv2.gain=vhz2.gain;

  sv1.calc(&sv1);              /* Call compute function for sv1 */
  sv2.calc(&sv2);              /* Call compute function for sv2 */

  x=sv1.va;                    /* Access the outputs of sv1 */
  y=sv1.vb;
  z=sv1.vc;

  p=sv2.va;                    /* Access the outputs of sv2 */
  q=sv2.vb;
  r=sv2.vc;

/* Do something with the outputs. Something is probably modulate PWMs to drive motors with.
*/

  }
```

Background Information

If the voltage applied to a three phase AC Induction motor is sinusoidal, then by neglecting the small voltage drop across the stator resistor, we have, at steady state,

$$\hat{V} \approx j\omega\ \hat{\Lambda}$$

i.e.,

$$V \approx \omega\ \Lambda$$

where $\hat{V}$ and $\hat{\Lambda}$ are the phasor representations of stator voltage and stator flux, and V and Λ are their magnitude, respectively. Thus, we get

$$\Lambda = \frac{V}{\omega} = \frac{1}{2\pi}\ \frac{V}{f}$$

From the last equation, it follows that if the ratio *V/f* remains constant for any change in *f*, then flux remains constant and the torque becomes independent of the supply frequency. In actual implementation, the ratio of the magnitude to frequency is usually based on the rated values of these parameters, i.e., the motor rated parameters. However, when the frequency, and hence the voltage, is low, the voltage drop across the stator resistor cannot be neglected and must be compensated for. At frequencies higher than the rated value, maintaining constant V/Hz means exceeding rated stator voltage and thereby causing the possibility of insulation break down. To avoid this, constant V/Hz principle is also violated at such frequencies. This principle is illustrated in Figure 10.

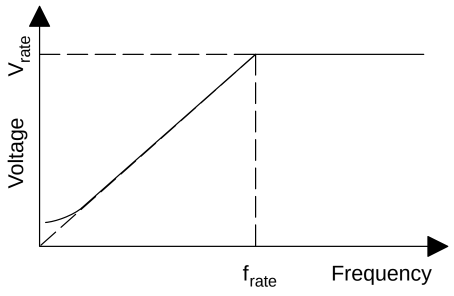


Figure 10. Voltage Versus Frequency Under the Constant V/Hz Principle

Since the stator flux is maintained constant (independent of the change in supply frequency), the torque developed depends only on the slip speed. This is shown in Figure 11. So by regulating the slip speed, the torque and speed of an AC Induction motor can be controlled with the constant V/Hz principle.

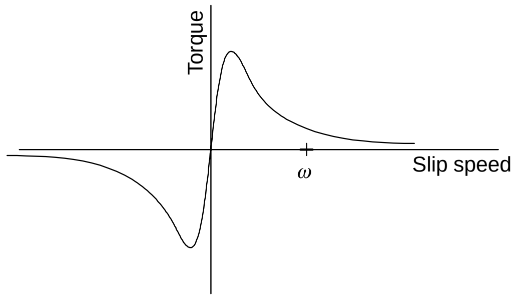


Figure 11. Toque Versus Slip Speed of an Induction Motor With Constant Stator Flux

Both open and closed-loop control of the speed of an AC induction motor can be implemented based on the constant V/Hz principle. Open-loop speed control is used when accuracy in speed response is not a concern such as in HVAC (heating, ventilation and air conditioning), fan or blower applications. In this case, the supply frequency is determined based on the desired speed and the assumption that the motor will roughly follow its synchronous speed. The error in speed resulted from slip of the motor is considered acceptable.

In this implementation, the profile in Figure 10 is modified by imposing a lower limit on frequency. This is shown in Figure 12. This approach is acceptable to applications such as fan and blower drives where the speed response at low end is not critical. Since the rated voltage, which is also the maximum voltage, is applied to the motor at rated frequency, only the rated minimum and maximum frequency information is needed to implement the profile.

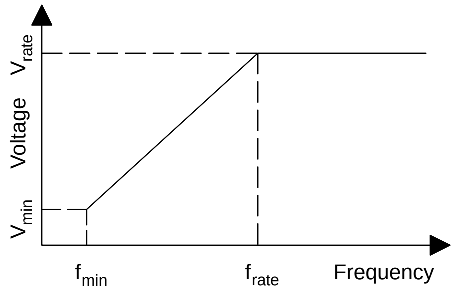


Figure 12. Modified V/Hz Profile

The command frequency is allowed to go below the minimum frequency, $f_{min,}$ with the output voltage saturating at a minimum value, V_{min}. Also, when the command frequency is higher than the maximum frequency, f_{max}, the output voltage is saturated at a maximum value, V_{max}.