ABSTRACT

ZHAONING, YANG. Digital Controller Design for Cascaded-Multilevel-Converter Based STATCOM Systems. (Under the direction of Alex Q. Huang.).

The purpose of the research has been to develop digital controllers for the Cascaded-Multilevel-Converter based STATCOM systems. STATCOM is one of the most important shunt FACTS controllers, which is designed to support the voltage and improve stability of the power system. Cascaded Multilevel Converter (CMC) is increasingly used at high power area due to its direct high voltage output with no need of transformer, which makes it as a good topology for STATCOM. Also because of the identical structure of each cell (H-bridge), CMC is the best candidate to be modularized.

In this thesis, two different digital controller developments are presented. First digital controller has conventional centralized controller structure. It uses a DSP plus FPGA as central controller. The DSP performs the control functions while FPGA is behaving like a bridge between DSP and peripheral devices. This topology is widely used in industry due to its simple and straightforward structure. Off-line simulation and hardware-in-the-loop real-time simulations are carried out to verify the design. Based on this topology a digital controller system has been developed and implemented in a three level STATCOM system.

However, the conventional centralized controller has some disadvantages. The central controller has direct connections with all converters. For high voltage and power rating converters, it is true that the power converters will consist of many modular blocks and these blocks will be placed at some distance from the controller. In this case, digital switch signals must be sent to the converters via optical fiber to improve reliability. However, the required fiber connections are too many that increase cost and the risk of fault. Moreover, the analog signals from the sensors such as the voltage and current are usually sending back to the

central controller through analog wire connections that has low electromagnetic interference (EMI) susceptibility.

To solve these problems, a modular digital controller topology is proposed. It has one central controller plus multiple local controllers. Every module converter has its own local controller which is placed very close to the modular converter. The local controller work as a “brain” for the converter. This “brain” realizes all the sensor signals of the module converter and sends them to central controller via asynchronous series communication protocol. Central controller performs the close loop algorithm and generates switching states. These switching states are sent to local controllers also through asynchronous series communication protocol. All the long distance data transmission is through optical links, which greatly increase the EMI susceptibility. The number of the fibers is reduced due to the series communication protocol. This modular controller is a little bit more complicated than the conventional centralized structure. But it can truly achieve “modularization”, which means improved reliability, isolation and increased expansion flexibility. This modular controller is built and verified through 50V 5A experiments.

Digital Controller Design for Cascaded-Multilevel-Converter Based STATCOM Systems

by

Zhaoning Yang

A thesis submitted to the Graduate Faculty of

North Carolina State University

In partial fulfillment of the

Requirements for the degree of

Master of Science

Electrical Engineering

Raleigh, NC

2006

Approved by:

_____	_____
Dr. Mo-Yuen Chow	Dr. Mesut Baran

_____

Dr. Alex Q. Huang
Chairman of Advisory Committee

To My Wife

Fang Yang

BIOGRAPHY

Zhaoning Yang was born in Yulin, Guangxi, China in 1978. He received his BS degree in Electrical Engineering in 1999 and MS degree in 2003 from Huazhong University of Science & Technology. Later, he entered Virginia Tech as a master student in Electrical Engineering in 2004. He had been working as a research assistant in Center for Power Electronics Systems for half a year. From 2004 to 2006, he has been with Semiconductor Power Electronics Center in North Carolina State University, where he has done research on digital controller development for STATCOM systems.

ACKNOWLEDGMENTS

I am grateful to my advisor, Dr. Alex Q. Huang, for his guidance and support throughout the work that led to this thesis as well as during the writing process. His motivating words and sharp direction has helped me in the completion of my work and research towards the completion of this thesis. None of this would have been possible without his support. I am also grateful to my committee members, Dr. Mesut Baran and Dr. Mo-Yuen Chow for help and suggestions throughout the research.

I would also like to thank Dr. Zhong Du, Wei Liu, Chong Han and Wenchao Song, with whom I have worked during the course of this research. I would like to extend a special thanks to Dr. Zhong Du, with whom I had wonderful discussions during this research. His keen knowledge and interest in power electronics applications and digital control systems was an inspiration. I often looked up to him for problems that I faced during my research.

I would like to extend my special thanks to Bin Chen, Yu Liu, Tiefu Zhao and Ding Li with whom I spent a great time during the course of my Masters. The friendly discourses were always welcome refreshment, and their cooperation was a source of motivation.

I also thank Dr. Subhashish Bhattacharya for over seeing my work and providing vital advice and guidance.

The great work of Mr. Jinseok Park in helping everyone in the lab with problems related to PCs and in always making everyone laugh with his great jokes also is worth appreciating.

I would like to extend a special thanks to all students at SPEC for their support and warmness. The friendly atmosphere in the lab was a really important and comforting feeling, and much

goes to the family-like group of students at SPEC.

Mr. Jerry Kirk made life at SPEC very easy, by helping in many vital issues, and I thank him for his great work.

Finally, my heartfelt appreciation goes toward my wife, Fang Yang for her support and encouragement throughout my education.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

LIST OF TABLES

Chapter 1 Introduction and Literature reviews

1.1 STATCOM (STATic synchronous COMpensator)

A Flexible AC Transmission System (FACTS) is an ac transmission system incorporating power electronic-based or other static controllers which provide better power flow control and enhanced dynamic stability by control of one or more ac transmission system parameters (voltage, phase angle. and impedance). In general, FACTS controllers can be divided into three categories [1]:

1. Series controllers
2. Shunt controllers
3. Combined series-shunt controllers

Among FACTS controllers, the shunt controllers have shown feasibility in term of cost effectiveness in a wide range of problem-solving applications from transmission to distribution levels. For decades, it has been recognized that the transmittable power through transmission lines could be increased, and the voltage profile along the transmission line could be controlled by an appropriate amount of compensated reactive current or reactive power. Moreover, the shunt controller can improve transient stability and can damp power oscillation during a post-fault event. Using a higher-speed power converter, the shunt controller can further alleviate or even eliminate the flicker problem.

The shunt controller basically consists of three groups.

1. Static var compensator (SVC)
2. Static synchronous compensator (STATCOM)
3. synchronous generator (SSG) or STATCOM with energy-storage system (ESS)

STATCOM is one of the most important shunt FACTS controllers, which have broad applications in electric utility industry. STATCOM has played an important role in power industry since 1980s [2] and recognized to be one of the key technologies in future power system. STATCOM is based on the principle that a voltage source inverter generates a controllable AC voltage source behind a reactance so that the voltage difference across the reactance produces active and reactive power exchange between the STATCOM and the transmission network line in a similar manner of a synchronous condenser, but much more rapidly. Fig. 1.1 shows the schematic configuration of STATCOM.

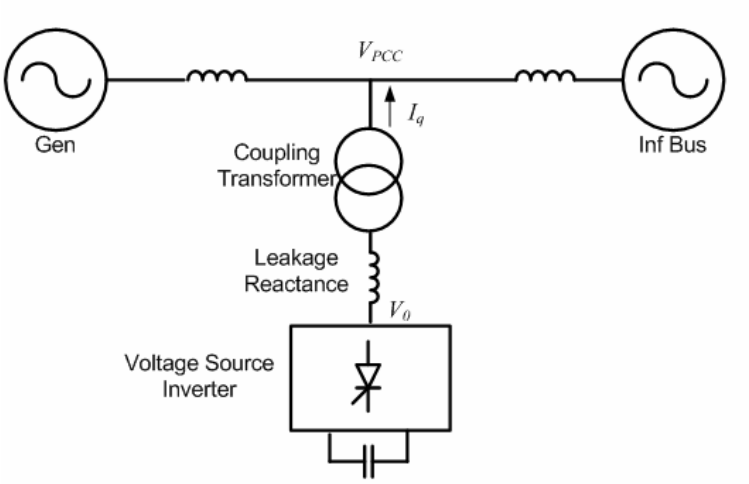


Figure 1.1. Schematic configuration of STATCOM.

Compared to SVC and other conventional reactive power compensators, STATCOM has several advantages:

a) STATCOM has the ability to maintain full output current range even at a very low system voltage. STATCOM uses a converter based var generator and functions as a shunt-connected synchronous voltage source. This is fundamentally different from SVC, which, with the thyristor-controlled reactors and thyristor-switched capacitors, functions as a shunt-connected, controlled reactive admittance. This basic operation difference (voltage source versus reactive admittance) accounts for the STATCOM's overall superior functional

characteristics, performance and greater application flexibility than SVC. Figure 1.2 shows the V-I characteristics of SVC and STATCOM. It shows that the STATCOM has the ability to maintain full output current range even at very low system voltage, which also makes it more effective than SVC in improving the transient stability. In contrast to STATCOM, the SVC's output current decrease linearly with the system voltage because SVC becomes a fixed capacitive admittance at full output.

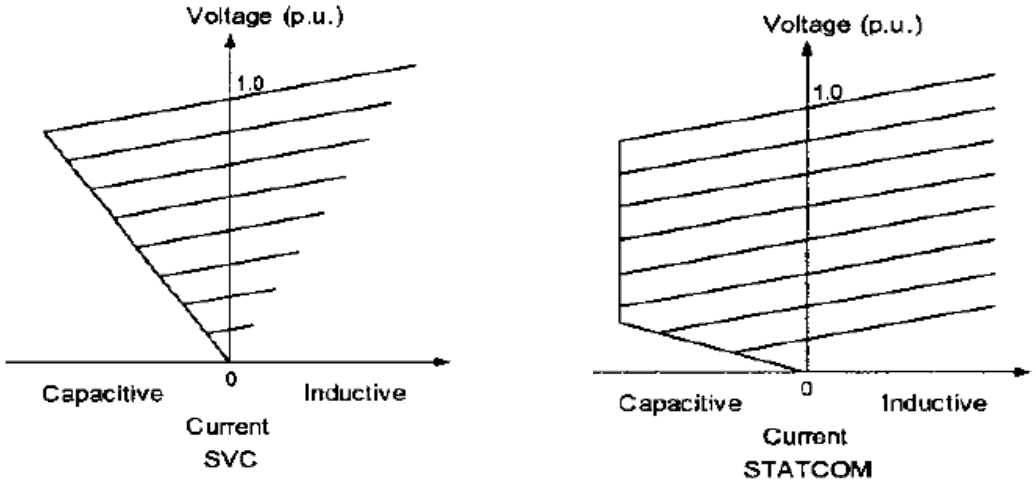


Figure 1.2. Comparison between V-I characteristics of SVC and STATCOM [5].

b) STATCOM has a dynamic performance far exceeding the other var compensators. The overall system response time of STATCOM can reach l0 ms or less [3]. SVC has thyristor controlled reactor and thyristor switched capacitor. Limited by having no turn-off capability of thyristor, the switching frequency is greatly reduced, which leads to a lower bandwidth and longer response time of SVC.

c) Capability to exchange real power: Because of the use of VSC, STATCOM can exchange real power with grid bi-directionally, and independently control both reactive power and real power, which is not realizable if using a SVC. Of course, this requires the DC

bus side of the STATCOM to be connected to an energy storage device or with another power source.

1.2 Cascaded Multilevel Converter

To implement STATCOM at meaningful MVA level, high power voltage source converter (VSC) is needed that in most cases exceeds the power handling capability of a simple two level converter without device series connection. One solution to this problem is to use multilevel VSC to increase the output voltage. Multilevel VSCs are emerging as a new breed of power converter options for high-power applications. The multilevel VSC typically synthesize the staircase voltage wave from several levels of dc capacitor voltages. One of the major limitations of the multilevel converters is the voltage unbalance between different levels. The techniques to balance the voltage between different levels normally involve voltage clamping or capacitor charge control.

In the past two decades, several multilevel VSC topologies have been reported and studied, i.e., diode clamped, flying capacitor and cascaded-multilevel converter (CMC) [4] topologies.

Figure 1.3 shows a five-level diode-clamped multilevel converter (DCMC) in which the dc bus consists of four capacitors. For dc-bus voltage V_{dc}, the voltage across each capacitor is V_{dc} /4 and each device voltage stress will be limited to one capacitor voltage level through clamping diodes. With different switching combinations, multilevel voltages can be synthesized. For voltage level Vdc, turn on four upper switches; for voltage level ¾ Vdc, turn on S_{a2}, S_{a3}, S_{a4}, $S_{a'1}$; for voltage level 1/2 Vdc, turn on S_{a3}, S_{a4}, $S_{a'1}$, $S_{a'2}$; for voltage level 1/4 Vdc, turn on S_{a4}, $S_{a'1}$, $S_{a'2}$, $S_{a'3}$; ; for voltage level 0 Vdc, turn on the lower four switches.

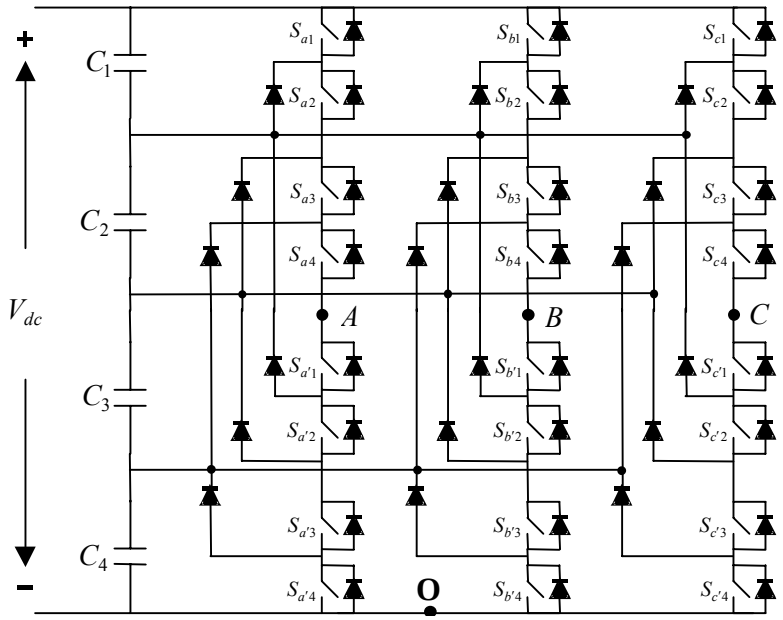


Figure 1.3. Three-phase, five-level diode-clamped converter.

Figure 1.3 shows a five-level flying capacitor multilevel converter (FCMC) topology. The voltage level defined in the flying-capacitor converter is similar to that of the diode-clamp type converter. The voltage synthesis in a flying-capacitor converter has more flexibility than a diode-clamp converter. For voltage level Vdc, turn on all upper switches; for voltage level ¾ Vdc, there are three combinations: (a) S_{a1}, S_{a2}, S_{a3}, $S_{a'4}$, (b) S_{a4}, $S_{a'1}$, $S_{a'2}$, $S_{a'3}$, (c) S_{a1}, S_{a3}, S_{a4}, $S_{a'2}$. For voltage level ¼ Vdc, there are six combinations: (a) S_{a1}, S_{a2}, $S_{a'3}$, $S_{a'4}$, (b) S_{a3}, S_{a4}, $S_{a'1}$, $S_{a'2}$, (c) S_{a1}, S_{a3}, $S_{a'2}$, $S_{a'4}$, (d) S_{a1}, S_{a4}, $S_{a'2}$, $S_{a'3}$, (e) S_{a2}, S_{a4}, $S_{a'1}$, $S_{a'3}$, (f) S_{a2}, S_{a3}, $S_{a'1}$, $S_{a'4}$. For voltage level ¼ Vdc, there are three combinations: (a) S_{a1}, $S_{a'2}$, $S_{a'3}$, $S_{a'4}$, (b) S_{a4}, $S_{a'1}$, $S_{a'2}$, $S_{a'3}$, (c) S_{a3}, $S_{a'1}$, $S_{a'2}$, $S_{a'4}$. For voltage level 0, turn on all lower switches.

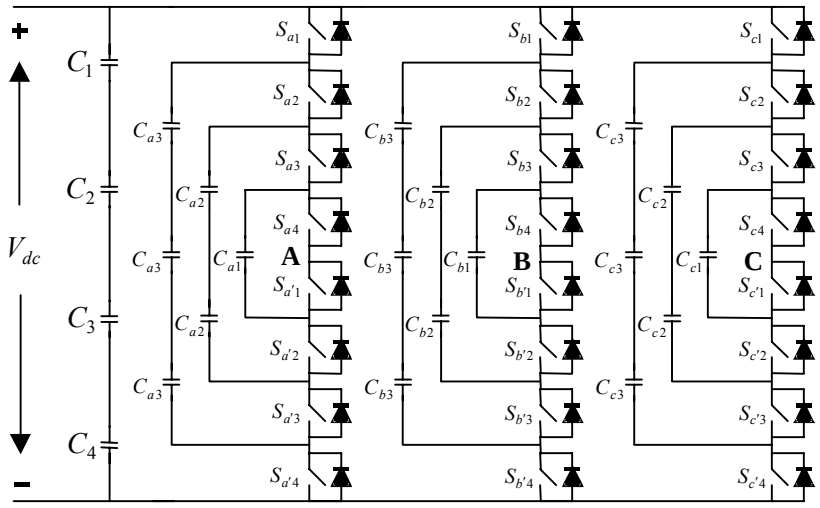


Figure 1.4. Three-phase, five-level flying capacitor converter.

Cascaded Multilevel Converter (CMC) is increasingly used at high power area due to its direct high voltage output with no need of transformer. Figure 1.5 shows the topology of a CMC. The "level" in a cascaded-inverters based converter is defined by m=2N+l, where m is the output phase voltage level and N is the number of dc sources. For example, a 7-level cascaded-inverters based converter will have three H-bridges. Compared to diode-clamped multilevel converters and flying capacitor multilevel converters, CMC is easy to design and assemble because of the uniform circuit structure of the converter units. Modularized circuit layout and packaging is possible in CMC topology, because each level has the same structure, and there are no extra clamping diodes or voltage-balancing capacitors, which are required in the DCMC and the FCMC. The number of output voltage levels can then be easily adjusted by changing the number of full-bridge converters.

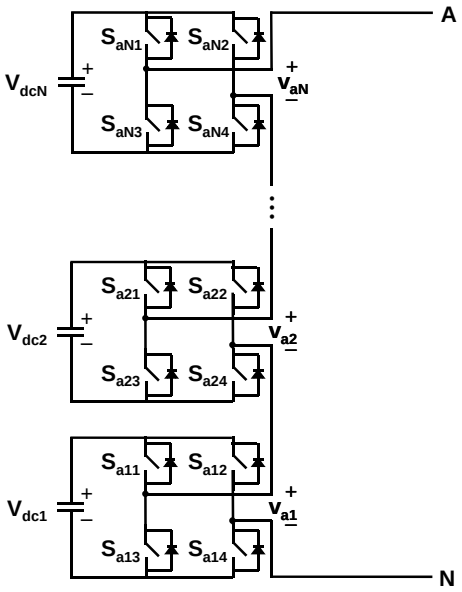


Figure 1.5. Cascaded Multilevel Converter with separated DC source

Table 1.1 [4] compares the main power component requirements per phase leg among these three multilevel VSCs, where m is the number of voltage levels. In Table 1.1, the number of main switches and main diodes needed by each converter to achieve the same number of voltage levels is the same. But DCMC (diode-clamp) needs extra clamping diodes, FCMC (flying-capacitor) needs extra balancing capacitors and CMC needs extra DC bus capacitors. So the total devices for each converter are different. To synthesize the same number of voltage levels, the CMC requires the least number of total main components.

Table 1-1. Comparison of power component requirements per phase leg among three converters.

Converter Type	Diode-clamp	Flying-capacitors	Cascaded-Multilevel
Main Switching devices	2(m-1)	2(m-1)	2(m-1)
Main diodes	2(m-1)	2(m-1)	2(m-1)
Clamping diodes	(m-1)(m-2)	0	0
DC bus capacitors	(m-1)	(m-1)	2(m-1)
Balancing capacitors	0	(m-1)(m-2)/2	0
Total	m^2+2m-3	$(m^2+8m-8)/2$	(9/2)(m-1)

1.3 Review of the Development of Digital Signal Processor

DSPs were first developed by Texas Instruments in the early 1980's for two major applications, i.e. military and speech analysis and synthesis. In both applications discrete Fourier transforms are essential. Hence, fast processing of additions and multiplications were a critical requirement. In 1980, the fastest processors available in personal computers (Motorola 68000 and Intel 8086) were not able to handle these tasks in real-time. Despite the integration of co-processors, the tremendous increase of clock speed and the introduction of

RISC architectures in general purpose processors, DSPs have been able to compete successfully (i.e. cost effectively) in real-time applications.

Till 1980, control hardware was mostly analog and could not handle the complicated algorithms which prevented widespread use of high-dynamic systems. The introduction of a signal processor (DSP) by Texas Instruments in 1982 almost eliminated overnight this impasse. In less than five years, industrial high-performance ac drives were built based on DSPs in Europe, Japan and the USA [7].

Due to its high capability to handle large-scale mathematic calculations, fast operating speed and interrupt driven response, the DSP has been widely used from the first day of its appearance. With the great advances in the microelectronics and VLSI technology, high-performance DSP can be effectively used to realize advanced control schemes in the motor drive, inverter, converter and many embedded systems. Most instructions of a DSP can be accomplished within one instruction cycle and complicated control algorithms can be realized by using software.

Thanks to the advances in semiconductor technology, the cost of DSP continues to drop while the performance keeps rising, as shown in Table 1.2 [8]. As the process technology

advances, the trend will continue.

Table 1-2. Two decades of DSP market integration [8].

	1982	1992	2002
Die Size (mm)	50	50	50
Technology Size (Microns)	3	0.8	0.18
MIPS	5	40	5,000
MHz	20	80	500
RAM (Words)	144	1,000	16,000
ROM (Words)	1,500	4,000	64,000
Power dissipation (mW/MIPS)	150	12.5	0.1
Transistors	50,000	500,000	5 million
Price (dollars)	150	15	1.50

1.4 Object of this work

This thesis is trying to show the developments of typical digital control system for the CMC based STATCOM. The thesis contains the objective of the STATCOM development, control algorithm, hardware design and software implementation.

1.5 Thesis Outline

The goal of this dissertation is to describe detailed research and development in achieving high-performance and reliable digital controllers for the CMC-based STATCOM. Major contributions are addressed as follows:

- Digital controller hardware & software design for a 4.5 MVA three-level CMC-based STATCOM
- Individual DC-Voltage loop to balance each module Voltage Source Converter's DC voltages
- Digital Modular controller prototype design for seven-level CMC STATCOM

Chapter 1 introduces the motivation and background of this thesis. High power converter topologies and applications are briefly described.

Chapter 2 is the review of modeling and control of CMC-based STATCOM. Here introduces procedures to model a CMC-based STATCOM system from electrical switching model to small signal model step by step. Finally, small signal transfer functions are achieved, which can be used to design the compensator for the system.

Chapter 3 shows the detail development of the digital controller for a 4.5 MVA STATCOM system. Firstly, the 4.5 MVA STATCOM system power stage and ETO devices are introduced. Secondly, Scale down experiment parameters and off-line simulation results and DSP on-line simulation results are described. Thirdly, digital controller hardware design is introduced. This section includes the selection of controller hardware and description of DSP + FPGA architecture. The fourth section of this chapter denotes the controller software design, include DSP configuration setup, PI controller design, and filter design, digital PLL loop implementation and graphical Human Machine Interface design. The last section of this chapter describes the 480V 100A experiment results.

Chapter 4 mainly describes a new digital modular controller prototype for a CMC based STATCOM system. Compared with the controller in chapter three, this new controller uses optimized centralized topology, central controller + local controllers. Detail hardware design is described in the second section of this chapter, including Modulation FPGA, Sensor FPGA, and local controller design. The series communication protocol is also explained in details. The third section described the 50V 5A experiments for the controller verification. This section includes system introduction, compensator design and individual DC voltage loops

design. The experiment results verify the design of the new topology of the modular digital controller.

Chapter 5 is summery and conclusion. It briefly reviews the whole dissertation and proposes the future work.

Chapter 2 Review of Modeling and Control of CMC-based STATCOM

This chapter presents the modeling and control of a CMC-based STATCOM. A schematic of a CMC-based STATCOM is shown in figure 2.1. Basically, the STATCOM system is composed of three main parts: a multilevel-cascaded VSC with separated DC capacitors, the coupling inductors, and a controller. A multilevel-cascaded converter consists of a number of identical H-bridge converters, whose output terminals are connected in series. The output voltage is thus the summation of those H-bridge converters, i.e.,

$$V_{kn} = V_{k1} + V_{k2} + ...V_{kN}$$

Where k is the phase notation, and N is the number of H-bridge converters per phase.

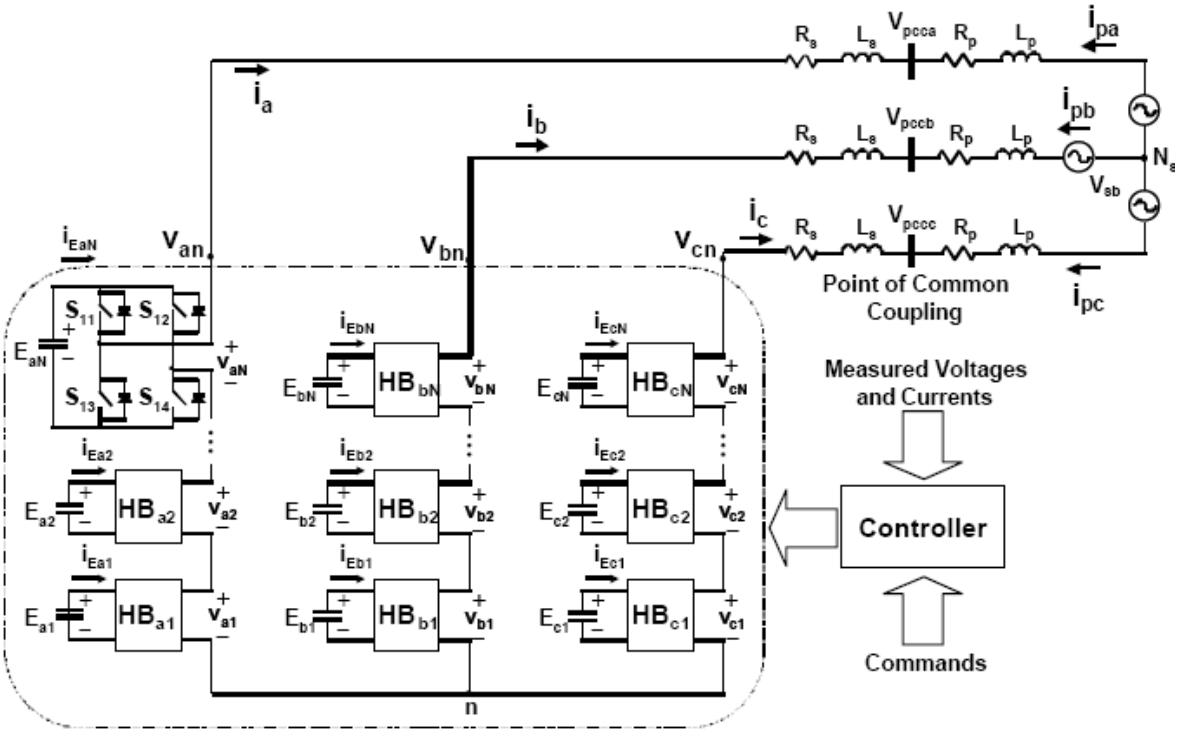


Figure 2.1. Schematic of a CMC-based STATCOM.

Since output voltage levels can be synthesized by an H-bridge converter, the total

number of output-phase voltage levels equals 2N+1. The maximum number of line-to-line voltage levels is 4N+1. Fig. 2.2 shows the synthesized phase voltage waveform of a CMC which has *N* H-bridge per phase.

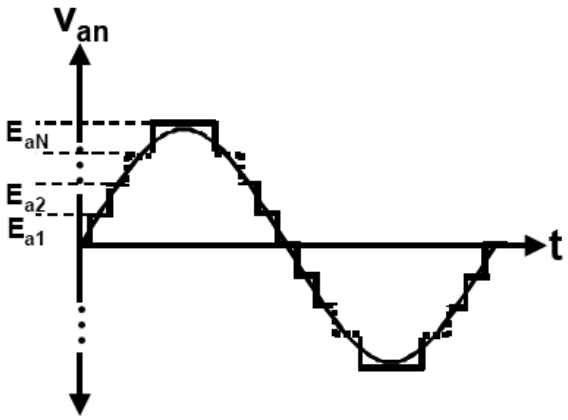


Figure 2.2. Phase voltage of a 2N+1 level CMC.

Fig. 2.3 shows a one-line diagram of a CMC-based STATCOM. The exchange of the real power and reactive power between the cascaded converter and the power system can be controlled by adjusting the amplitude and phase of the converter output voltage. In case of ideal converter, the output voltage of the converter is controlled to be in phase with that of the power system. To operate the STATCOM in capacitive mode, the magnitude of the converter output voltage is greater than that of the power system. On the other hand, the magnitude of the output voltage of the converter is controlled to be less than that of the power system to absorb reactive power or to operate the STATCOM in inductive mode. However, in practice, the converter associates with internal losses caused by non-ideal power semiconductor devices and passive components. As a result, without any control, the capacitor voltage will decrease. To regulate the capacitor voltage, a small phase shift between the converter voltage and the power system voltage is introduced.

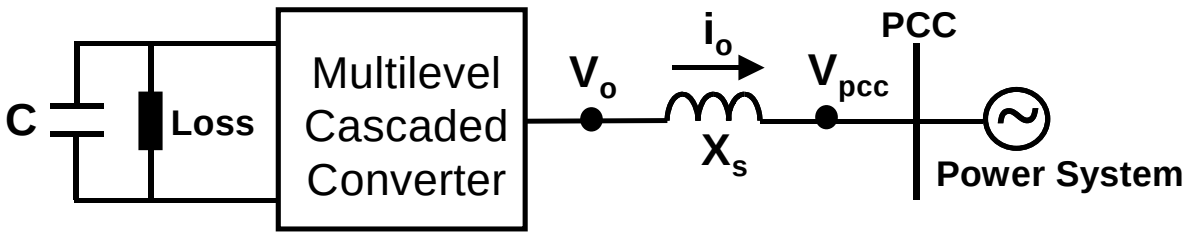


Figure 2.3. One-Line diagram of a CMC-based STATCOM.

Fig. 2.4 shows the CMC model development procedure from electrical abc coordinate model to average dq0 coordinate model [9]. The process is begun by defining the switching functions of the converter power stage. By applying the defined switching functions, the CMC can be represented by a switching model. The next step is to average the switching model. In this step, all switching behaviors are neglected; only fundamental components are considered. A generalized average model is derived in abc coordinates and is shown in figure 2.5.

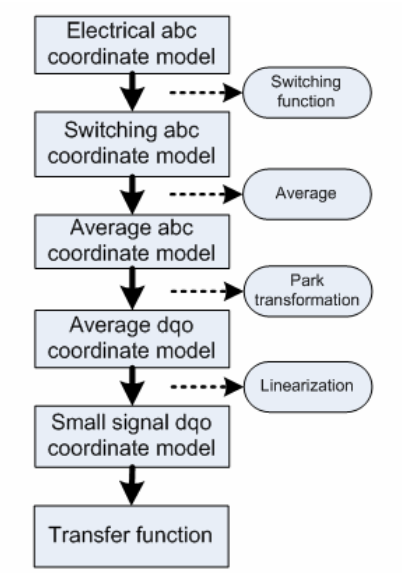


Figure 2.4. Model development of CMC-based STATCOM

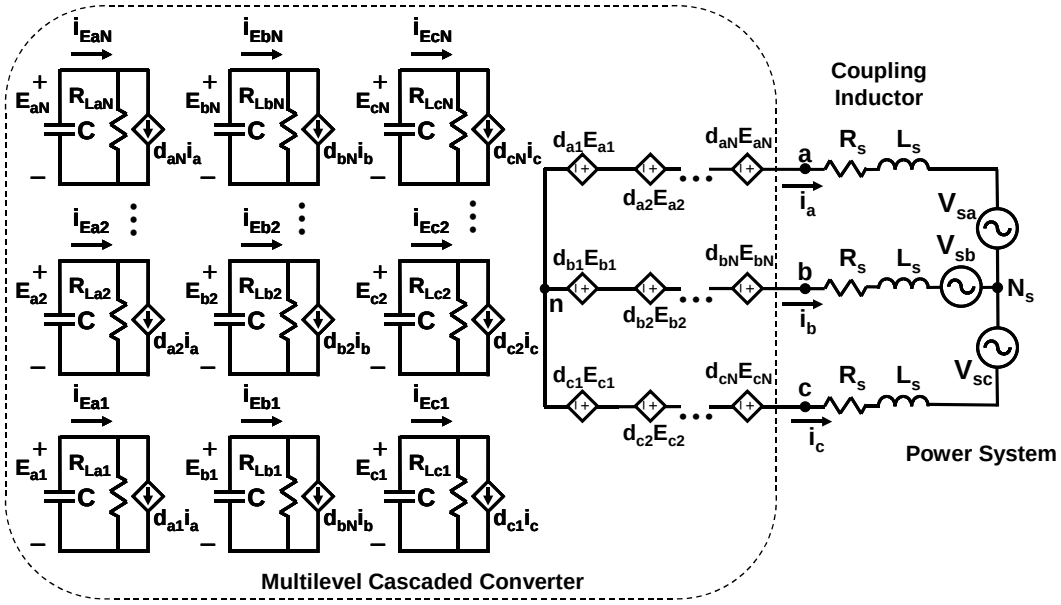


Figure 2.5. Average model of CMC-based STATCOM in abc coordinate [9].

A further simplified average model in abc reference frame can be achieved by assuming:

All dc voltage sources are assumed to be equal.

The losses in each H-bridge converter in the same level are identical.

Due to equal synthesized output voltages, all duty cycles in the same phase are assumed to be identical, i.e.,

$d_k = d_{k1} = d_{k2} = \cdots = d_{kN}$, where $k = a, b, c$

This simplified abc average model is shown in Fig. 2.6.

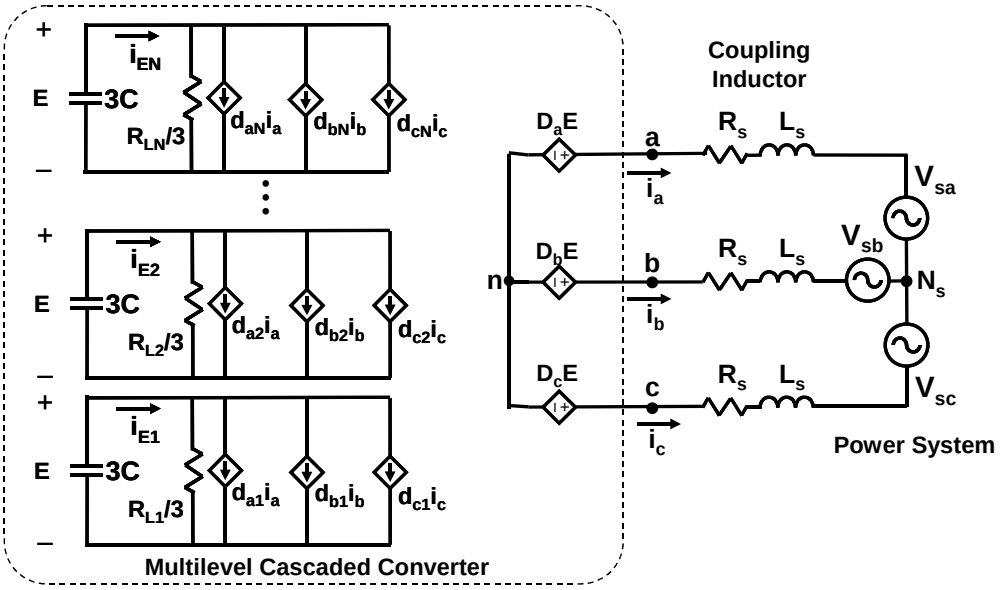


Figure 2.6. Simplified average model for CMC STATCOM in abc frame [9].

From the average model in abc-frame as shown in Fig. 2.6, two characteristic equations can be derived:

$$\frac{d\vec{i}_{abc}}{dt} = \frac{E \cdot \vec{D}_{abc}}{L_s} - \frac{\vec{v}_{sabc}}{L_s} - \frac{R_s}{L_s}\vec{i}_{abc} \qquad \text{Equation 2-1}$$

$$\frac{dE}{dt} = \frac{-3E}{R_{Lj}C} - \frac{\vec{d}_{abc}{}^{T} \cdot \vec{i}_{abc}}{3C} \quad , j = 1...N \qquad \text{Equation 2-2}$$

$$\text{Where, } \vec{i}_{abc} = \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix},\ \vec{D}_{abc} = \begin{bmatrix} D_a \\ D_b \\ D_c \end{bmatrix},\ \vec{d}_{abcj} = \begin{bmatrix} d_{aj} \\ d_{bj} \\ d_{cj} \end{bmatrix},\ \vec{v}_{sabc} = \begin{bmatrix} v_{sa} \\ v_{sb} \\ v_{sc} \end{bmatrix}.$$

Figure 2.7 shows the phasor diagram of DQ0 coordinate. Where DQ0 coordinate rotate at the same angle velocity of that of $V_{pcc}(t)$. $V_{pcc}(t)$ is inline with D axis. Then vector $V_{pcc}(t)$ becomes stationary in the DQ0 space. In other words, the behavior of balanced, three-phase, time-variant parameters in the ABC coordinates can be simply represented by two time-invariant parameters in the DQ0 coordinates. Park's transformation matrix (equation 2-3), is a tool for transferring parameters in the ABC into DQ0 coordinates.

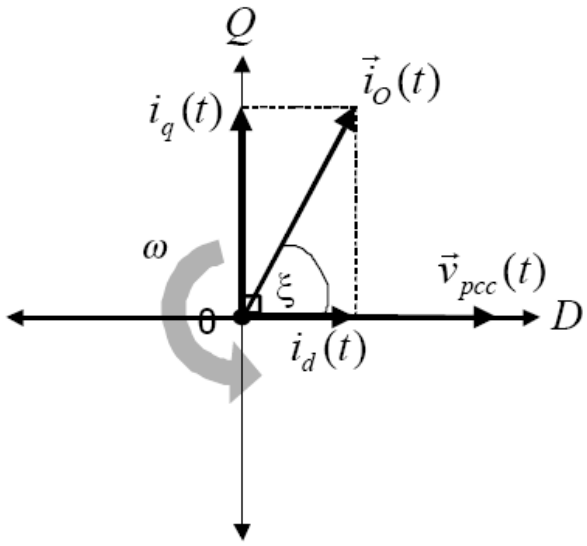


Figure 2.7. Phasor diagram of the DQ0 coordinate

$$\vec{T}_{dq0/abc} = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\theta) & \cos(\theta - \frac{2\pi}{3}) & \cos(\theta + \frac{2\pi}{3}) \\ -\sin(\theta) & -\sin(\theta - \frac{2\pi}{3}) & -\sin(\theta + \frac{2\pi}{3}) \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \qquad \text{Equation 2-3}$$

Where $\theta = \int_0^t \omega(\tau) d\tau + \theta(0)$, $\omega(\tau)$ is the angular velocity.

By applying this rule, the parameters in the DQ0 coordinates in line with the D-axis provide real components, whereas the Q-axis components represent the reactive components. Therefore, the relationship between the instantaneous three-phase power components in the DQ0 coordinates can be expressed in Equation 2-4 and 2-5. The amount of transferred reactive power can be controlled by adjusting the magnitude of the converter output voltage, and the amount of transferred real power can be controlled by adjusting the phase of the converter output voltage. Therefore the real power and reactive power exchange between the STATCOM and power grid can be controlled independently by control the D-axis and Q-axis output current of the STATCOM.

$$P(t) = 2 * \left|V_{pccd}\right| \cdot \left|I_d\right| = 2 * \left|\vec{V}_{pcc}\right| \cdot \left|\vec{I}_o\right| \cos(\xi) \qquad \text{Equation 2-4}$$

$$Q(t) = -2 * \left|V_{pccd}\right| \cdot \left|I_q\right| = -2 * \left|\vec{V}_{pcc}\right| \cdot \left|\vec{I}_o\right| \sin(\xi) \qquad \text{Equation 2-5}$$

Multiply equation (2-1) and (2-2) by park's transformation matrix (equation 2-3) in both side, we can achieve the average model in dq0 reference frame, as depicted at Fig 2.8.

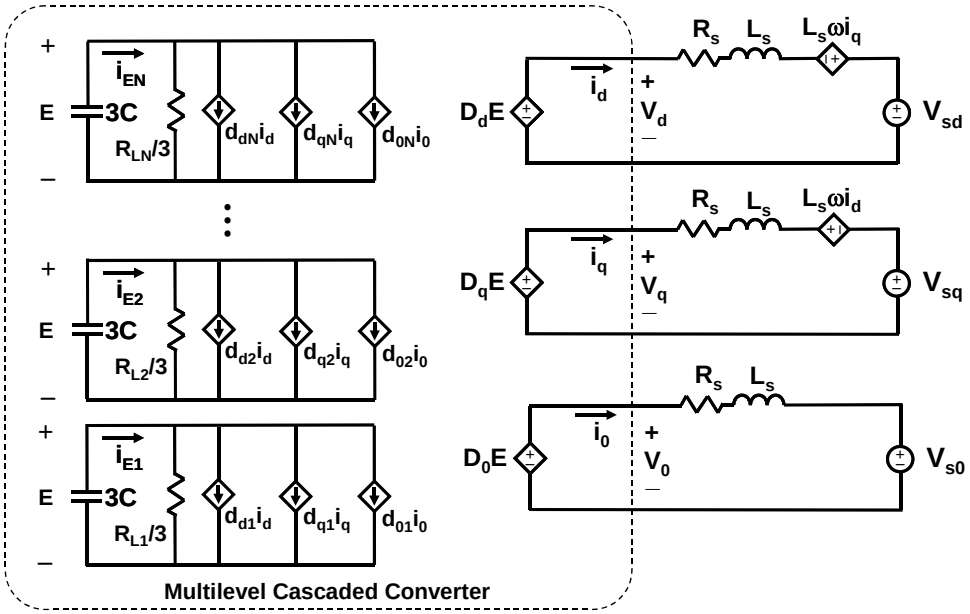


Figure 2.8. Simplified average model for CMC STATCOM in dq0 frame [9].

The equation in dq0 reference frame:

$$\frac{d}{dt}\begin{bmatrix} i_d \\ i_q \\ i_0 \end{bmatrix} = \frac{E}{L_s}\begin{bmatrix} D_d \\ D_q \\ D_0 \end{bmatrix} - \frac{1}{L_s}\begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{s0} \end{bmatrix} - \begin{bmatrix} \frac{R_s}{L_s} & -\omega & 0 \\ \omega & \frac{R_s}{L_s} & 0 \\ 0 & 0 & \frac{R_s}{L_s} \end{bmatrix} \cdot \begin{bmatrix} i_d \\ i_q \\ i_0 \end{bmatrix} \qquad \text{Equation 2-6}$$

$$\frac{dE}{dt} = \frac{-3E}{R_{Lj}C} - \frac{1}{3C}[d_{dj} \quad d_{qj} \quad d_{0j}] \cdot \begin{bmatrix} i_d \\ i_q \\ i_0 \end{bmatrix} \qquad \text{Equation 2-7}$$

A small signal model can be developed by linearizing all variables in (2-6) and (2-7) around their quiescent operating points. One gets:

$$\frac{d}{dt}\begin{bmatrix}\tilde{i}_d\\ \tilde{i}_q\\ \tilde{i}_0\end{bmatrix}=\frac{E}{L_s}\begin{bmatrix}\tilde{D}_d\\ \tilde{D}_q\\ \tilde{D}_0\end{bmatrix}+\frac{\tilde{E}}{L_s}\begin{bmatrix}D_d\\ D_q\\ D_0\end{bmatrix}-\frac{1}{L_s}\begin{bmatrix}\tilde{v}_{sd}\\ \tilde{v}_{sq}\\ \tilde{v}_{s0}\end{bmatrix}-\begin{bmatrix}\frac{R_s}{L_s} & -\omega & 0\\ \omega & \frac{R_s}{L_s} & 0\\ 0 & 0 & \frac{R_s}{L_s}\end{bmatrix}\cdot\begin{bmatrix}\tilde{i}_d\\ \tilde{i}_q\\ \tilde{i}_0\end{bmatrix} \qquad \text{Equation 2-8}$$

$$\frac{d\tilde{E}}{dt}=-\frac{3\tilde{E}}{R_{Lj}C}-\frac{1}{3C}\begin{bmatrix}I_d & I_q & I_0\end{bmatrix}\cdot\begin{bmatrix}\tilde{d}_{dj}\\ \tilde{d}_{qj}\\ \tilde{d}_{0j}\end{bmatrix}-\frac{1}{3C}\begin{bmatrix}D_{dj} & D_{qj} & D_{0j}\end{bmatrix}\cdot\begin{bmatrix}\tilde{i}_d\\ \tilde{i}_q\\ \tilde{i}_0\end{bmatrix} \qquad \text{Equation 2-9}$$

Where $j=1...N$, N is the number of H-bridge converters per phase;

$\tilde{i}$ is the small-signal AC variation of i, whose quiescent operating point is I;

$\tilde{d}$ is the small-signal AC variation of d, whose quiescent operation point is D.

Thus, the small signal model of the STATCOM system is depicted in Fig. 2.9.

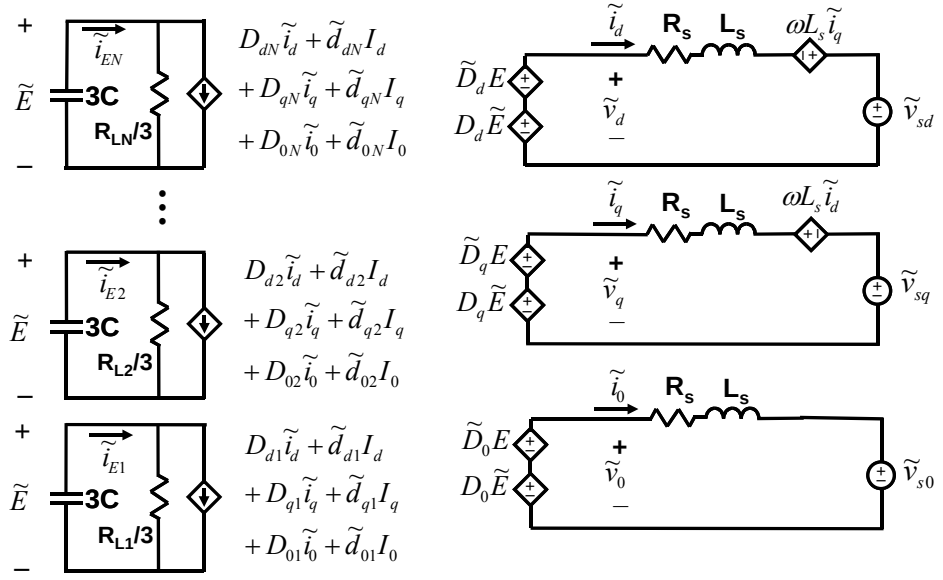


Figure 2.9. Small signal model for CMC STATCOM in dq0 frame [9].

From the small-signal model, the open-loop control-to-output current and control-to-dc bus transfer functions can be realized as shown in Fig. 2.10

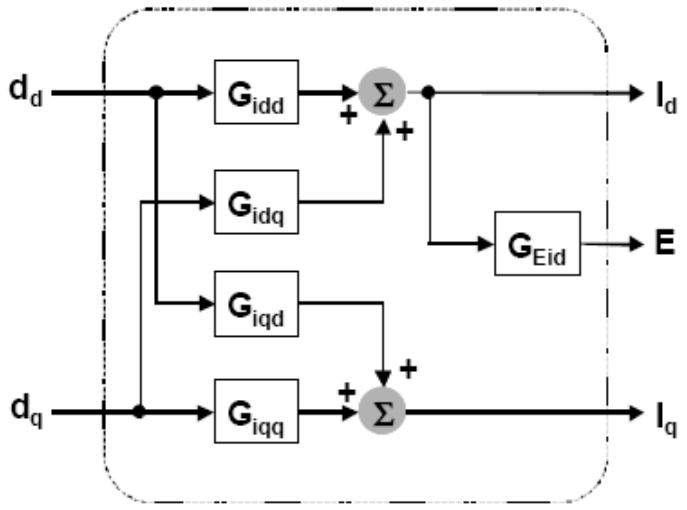


Figure 2.10. Open-loop transfer function [9]

$$G_{idd} = \frac{\tilde{i}_d}{\tilde{d}_d} = \frac{NE(R_s + L_s S)}{L_s{}^2 S^2 + 2L_s R_s S + R_s^2 + \omega^2 L_s^2} \qquad \text{Equation 2-10}$$

$$G_{iqq} = \frac{\tilde{i}_q}{\tilde{d}_q} = \frac{NE(R_s + L_s S)}{L_s{}^2 S^2 + 2L_s R_s S + R_s^2 + \omega^2 L_s^2} \qquad \text{Equation 2-11}$$

$$G_{idq} = \frac{\tilde{i}_q}{\tilde{d}_d} = \frac{-NE\omega L_s}{L_s{}^2 S^2 + 2L_s R_s S + R_s^2 + \omega^2 L_s^2} \qquad \text{Equation 2-12}$$

$$G_{iqd} = \frac{\tilde{i}_q}{\tilde{d}_d} = \frac{-NE\omega L_s}{L_s{}^2 S^2 + 2L_s R_s S + R_s^2 + \omega^2 L_s^2} \qquad \text{Equation 2-13}$$

$$G_{Eidj} = \frac{\tilde{E}_j}{\tilde{i}_d} = \frac{-(D_{dj} L_s S + D_{qj} \omega L_s + D_{dj} R_s)}{3CL_s S^2 + 3CR_s S + D_{dj} D_{qj} N} \qquad \text{Equation 2-14}$$

Based on this transfer function, the closed-loop control block diagram of a three level CMC-based STATCOM can be shown in the figure 2.11. The main objective of the feedback controller is to regulate the Q channel current following its command as fast as possible. Also

to maintain the DC bus voltages at the set point, an external loop H_{Ed} is presented to generate D channel current reference. The input *E* is the average three phase DC bus voltage. For more details please refer to [9] [10][11].

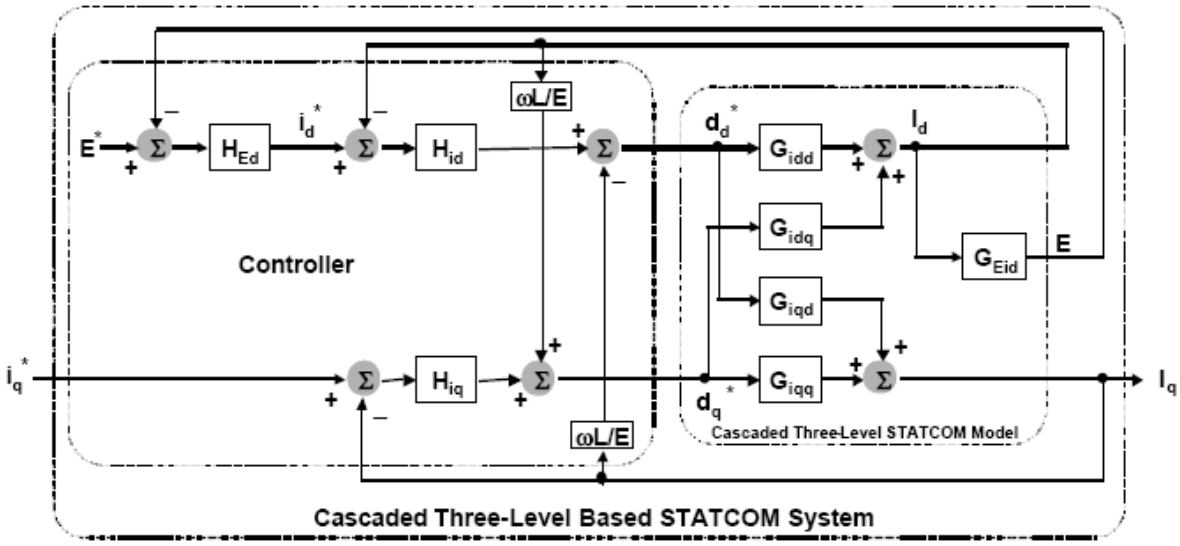


Figure 2.11. Closed-loop control block diagram of CMC-based STATCOM [9]

Chapter 3 Centralized Digital Controller for CMC-based STATCOM

This chapter described the centralized controller design for a three-level CMC-based STATCOM, including control strategy, controller hardware & software design and experiment results. A three-level conventional centralized controller topology is shown in Fig. 3.1. The centralized topology is very straight forward and simple. It's widely used in industry because of the easy implementation. There are some disadvantages of this structure:

1. Connections are complex between controller and power stage.
2. Low reliability.
3. Noise interference.
4. Low expansion flexibility.

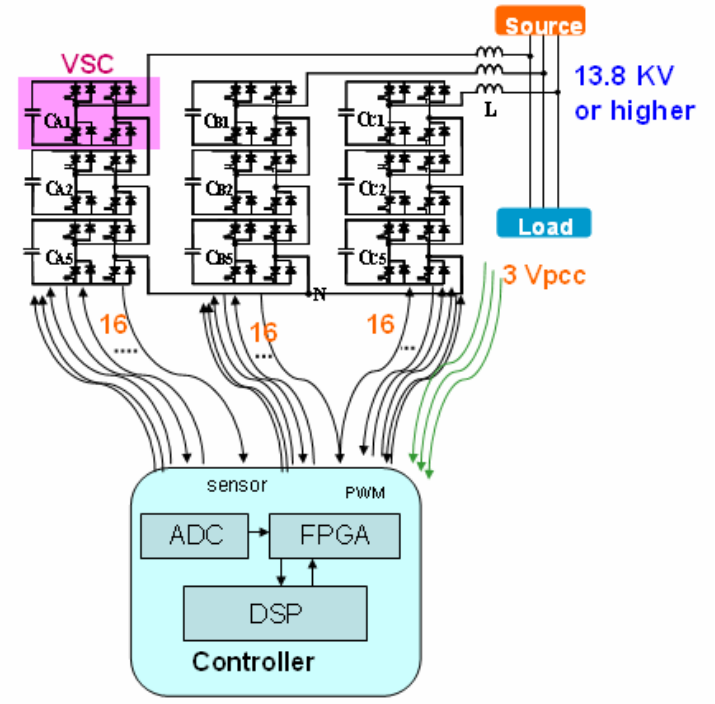


Figure 3.1 Centralized controller topology

3.1 STATCOM System Introduction

The CMC STATCOM system is based on three 1.5 MVA Emitter Turn-Off thyristor (ETO)-based modular VSC. The ETO is a promising semiconductor device for high switching frequency, high-power operations, which combines the advantages of thyristor's high voltage/current capability and MOSFET's (Metal-Oxide Semiconductor Field Effect Transistor) easy gate control. Due to very high silicon utilization and a greatly simplified drive circuit, its cost is much lower than other high-power competing technologies, such as GTO, IGCT, and IGBT. Figure 3.2 shows a generation-3 4.5 kV/4 kA ETO device and the 1.5 MVA ETO-based modular VSC prototypes. Three ETO-based 1.5MVA H-bridge converters has been built and tested. The designed DC bus voltage is 2.0 kV, and the output RMS current is l.0kA with the switching frequency of l.0 kHz [13] [14].

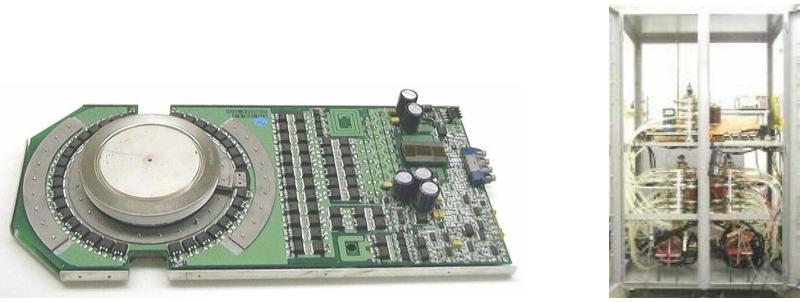


Figure 3.2 (a) Generation 3 4.5 kV/5 kA ETO device, (b) 1.5 MVA ETO-based modular VSC [13].

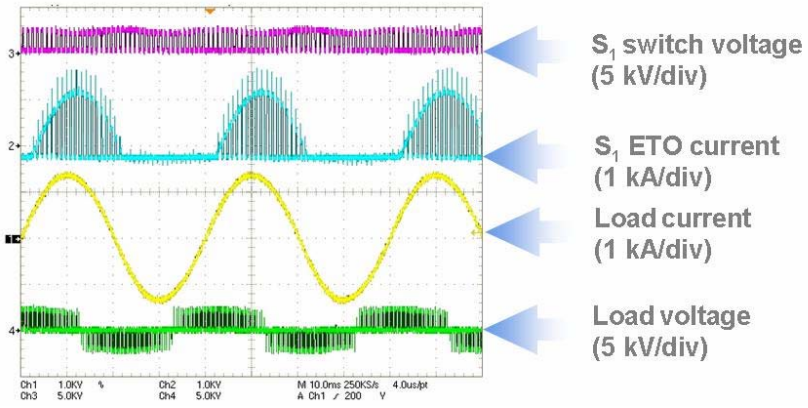


Figure 3.3. (b) Continuous power test [13]

The one-line diagram of the 4.5 MVA CMC STATCOM is shown in figure 3.4. The STATCOM system connected to the grid at a 480V feeder without transformer connection. A 500A circuit breaker is installed behind the AC source. It can be controlled by the central controller through optical fiber to disconnect the STATCOM from grid, in case of fault situation. To limit the inrush current during the STATCOM connect to AC grid, there is a pre-charge network.

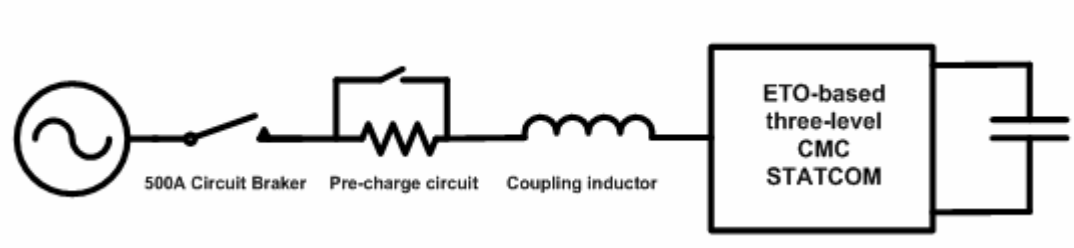


Figure 3.4. One-line diagram of 4.5 MVA three-level CMC STATCOM

To cool the converter system, a water-cooling system that consists of three water-to-water heat exchangers is connected to a condenser tower. A detailed system integration diagram including control center is shown in Figure 3.5.

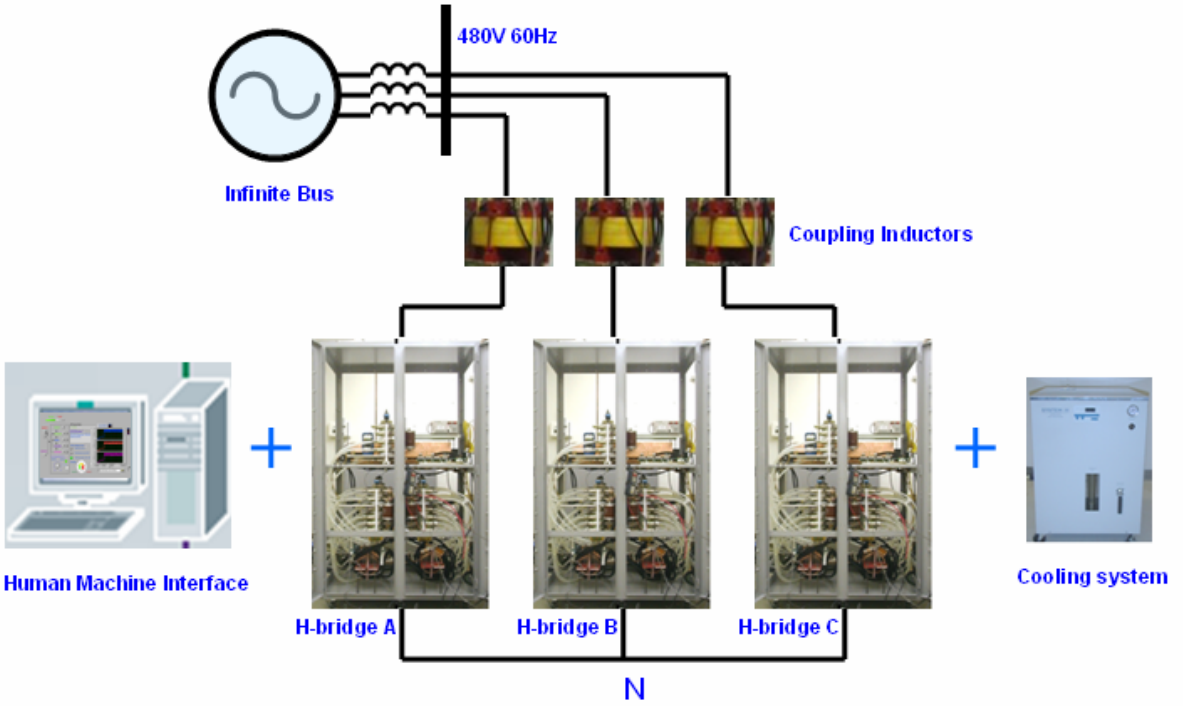


Figure 3.5. 4.5 MVA STATCOM system integration

According to the developed small signal dq0 model, the control strategy for a three-level (N=1) CMC-based STATCOM is shown in Fig. 3.6. The Phase Lock Loop (PLL) block generates angle information for Park transformation base on the transmission line phase voltages. Because it's a 3-level CMC converter, there is only one H-bridge in each phase. The DC voltage error, which is the difference between averaged three DC bus voltages E and reference DC bus voltage E_{ref}, is the input of DC voltage compensator H_E. H_E generates the d-axis reference current I_{dref} to maintain the DC-bus voltages. I_{qref} is the reference q-axis current, which is the control command of the reactive power output. When it is a negative value the STATCOM outputs reactive power (capacitive mode); when it's positive the STATCOM absorbs reactive power (inductive mode). H_{id} and H_{iq} are the current-loop compensators. The outputs of them are duty cycles in dq0 format. Assuming the AC system is a balanced 3-phase system, the 0-axis value is always 0. Through inverse Park

transformation, the duty cycles in ABC coordinate will be generated and feed into FPGA PWM generator to create switching signals.

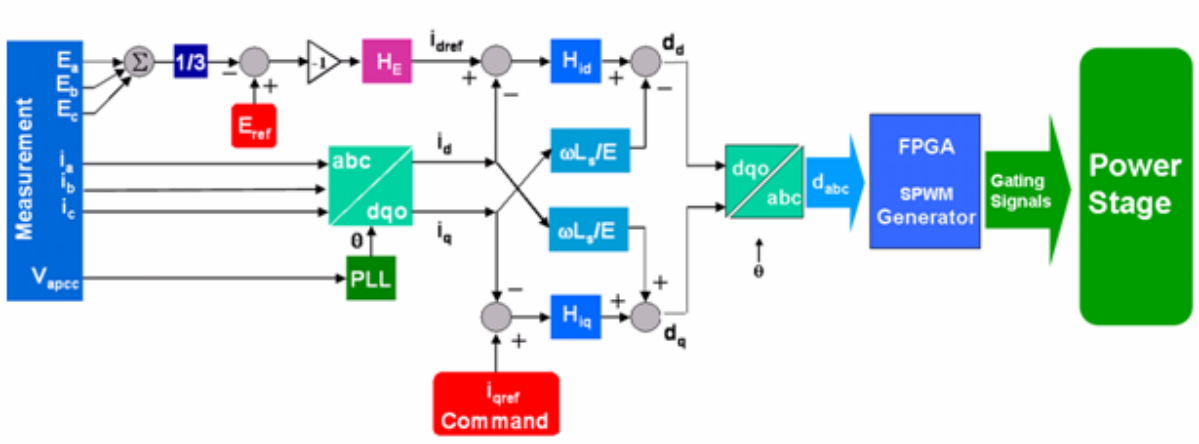


Figure 3.6. Control strategy block diagram of three-level STATCOM

3.2 STATCOM Digital Controller Hardware Design

The whole STATCOM control system hardware consists of four parts: central controller unit, sensor signal conditional board (Analog board), digital signal board, and power supply. In order to achieve fast dynamic response with robust performance, DSP + FPGA architecture is selected as the central control unit.

Figure 3.7 shows the overview of this controller hardware setup. The analog board is a sensor signal conditioning board. It scales down the sensor output signals within 1V and transfers them to A/D converters. Through A/D converters the FPGA realizes all the sensor signals. There is a parallel 32-bit External Memory Interface (EMIF) between DSP and FPGA. DSP performs the control algorithm and generates duty cycle signals. FPGA receives the duty cycle signals and generates the PWM signal. The PWM signals are sent to the digital board through a 40 pin flat cable. Digital board then transfers these signals to optical signals and send to each ETO devices.

Figure 3.7. 4.5 MVA STATCOM controller

3.2.1 Requirement on the DSP computation capacity

In this section, we discuss the criteria for selecting a proper DSP, why the architecture DSP+FPGA is a must and how it works. In the selection of a proper DSP, the performance and cost are always concerns. Compare with the high cost of the high power STATCOM power stage, cost of the controller is not a big problem. We just care about the performance and reliability. Usually, working speed and the word length are two dominant facts used to determine DSP's performance. First we study how to estimate the working speed requirement based on one certain task. Based on this estimation, we can get an approximate value of DSP computation capacity.

The first fact we have to consider is the max sampling/controller execution frequency, which will affect the control performance greatly. There are several items in the system, which have different requirements on the speed. Based on the synchronous machine we have, the variable with the highest frequency in the system is the Digital Phase Lock Loop (DPLL). The accuracy of the DPLL is extremely important. From the equation of the power calculation we know that even a very small error of the phase angle will lead to a huge error reactive power output of STATCOM. The error of the DPLL loop must be minimized.

Therefore we choose 20 kHz as the sample frequency for the DPLL loop which is over 300 times of the fundamental frequency. Therefore we need a 20 kHz Interrupt Service Routine (ISR) to finish all the algorithm calculation.

With the knowledge of the sampling /controller execution frequency, now we need to estimate the computation requirement of the control algorithm in one execution. We can calculate the total computation requirement for the DSP through a multiplication operation. In chapter 2.1, we draw a complete and clear picture about the control algorithm. We need to implement several control loops: DPLL, DC bus voltage regulation loop, current loop. This requires several transfer functions. Each regulation loop includes at least one PI controller and maybe need Low Pass Filter (LPF). Therefore these control loops will probably take 20 thousand instruction cycles. Also DSP need to read the sensor signals from FPGA through the parallel data bus. This process is pretty slow due to the great time consumption of waiting the A/D converter to finish conversion. And these digitized signals need to be calibrated again in DSP to retrieve their analog values. Sensor signals are two Line-Line V_{pcc} voltages, three phase currents and three phase DC bus voltages. These process cost a lot of time, assume 20 thousands instruction cycles.

Given the assumption that the sampling/controller execution frequency is 20 kHz and 40000 instructions are needed per execution cycle, the instruction speed should be at least 800 (20000× 40000) million instructions per second (MIPS). Based on this value, we can choose a proper DSP. A 32-bit TMS320c6000 series DSP platform from Texas Instruments can satisfy our need. A TMS320c6713 DSP Starter Kit (c6713DSK) is chosen as our platform. It include a TMS320c6713 225 MHz 32-bit floating points DSP, which can process 1800 million floating point instructions per second. This powerful and reliable DSP platform

not only satisfies the computation requirements for 3-level STATCOM but also has enough capacity for the future upgrade, for example 13-level STATCOM.

Figure 3.8 shows the block diagram of c6713DSK board. Key features include:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz
- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)
- Software board configuration through registers implemented in CPLD
- Configurable boot options
- Standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator.

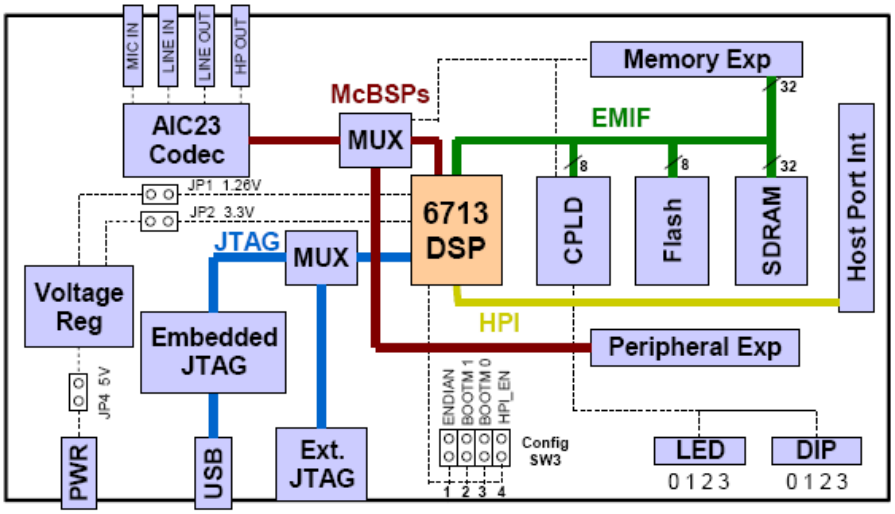


Figure 3.8. Block diagram of c6713DSK board [16]

3.2.2 Description of the architecture of DSP+FPGA

This c6713DSK board is very powerful, high performance platform and is widely used for audio, graphics and communication applications. In those kinds of applications mentioned above, the main purpose of the DSP is to process large quantity of data. But in the

control application, except the data processing, DSP needs to do some discrete I/O control with peripheral devices. Unlike other series DSP starter kit, for example TMS320c2400 series, which is mainly designed for motor control, c6713DSK itself doesn't have A/D converter and Pulse Width Generator. So naturally, one DSP plus FPGA architecture becomes popular, especially for the control purpose. In this architecture, the DSP is dedicated to do the data processing and control execution, while the FPGA is behaving like a bridge between DSP and peripheral devices. FPGA is focusing on the I/O communication and some preliminary data processing or even taking a large part of data processing work given a high performance FPGA.

With the help of the 32-bit EMIF connector and peripheral expansion connector we are able to plug a daughter board. This daughter board is developed by SIGNALWARE Corporation, which contains a Xilinx XCV300 300,000 gates FPGA. XCV300 is a high performance FPGA developed by Xilinx for the high-density application. Moreover there are 4 A/D chips on the daughter board, which provide 16 A/D channels and they could be sampled up to 6MS/s with 12 bits. Figure 3.9 shows the block diagram of AED-106 daughter board.

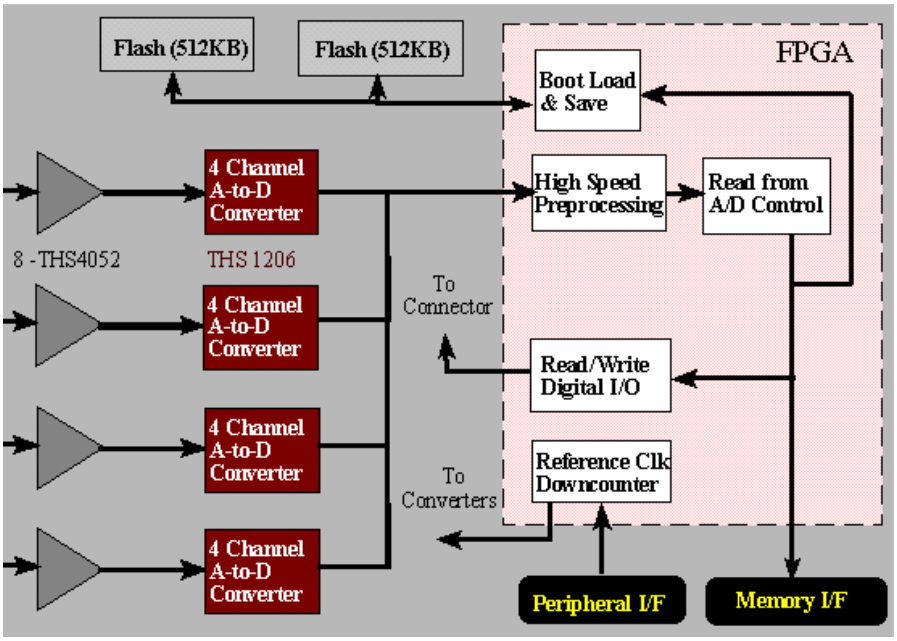


Figure 3.9. AED-106 daughter board block diagram [17]

This daughter board also provides 24 digital I/Os. They can be configured to be input and output. Figure shows the I/O configuration of the central controller units. We use 12 digital I/Os as main switch gate drivers. Use 3 digital I/Os to control the auxiliary switches. The auxiliary switches include AC switch, DC circuit breaker and pre-charge switch.

As we discussed above we need to use eight A/D channels to sense the analog signals, include two Line-Line voltage signals, three-phase converter currents and three DC bus voltages. We also use one analog channel as the “Fault” signal input. When we push the emergency stop button or there is a fault gate driver signal which has been detected by the digital board, a “Fault” signal is generated and sent to this channel. Then the DSP will shut down the STATCOM. Figure 3.10 shows the I/O of the central control unit.

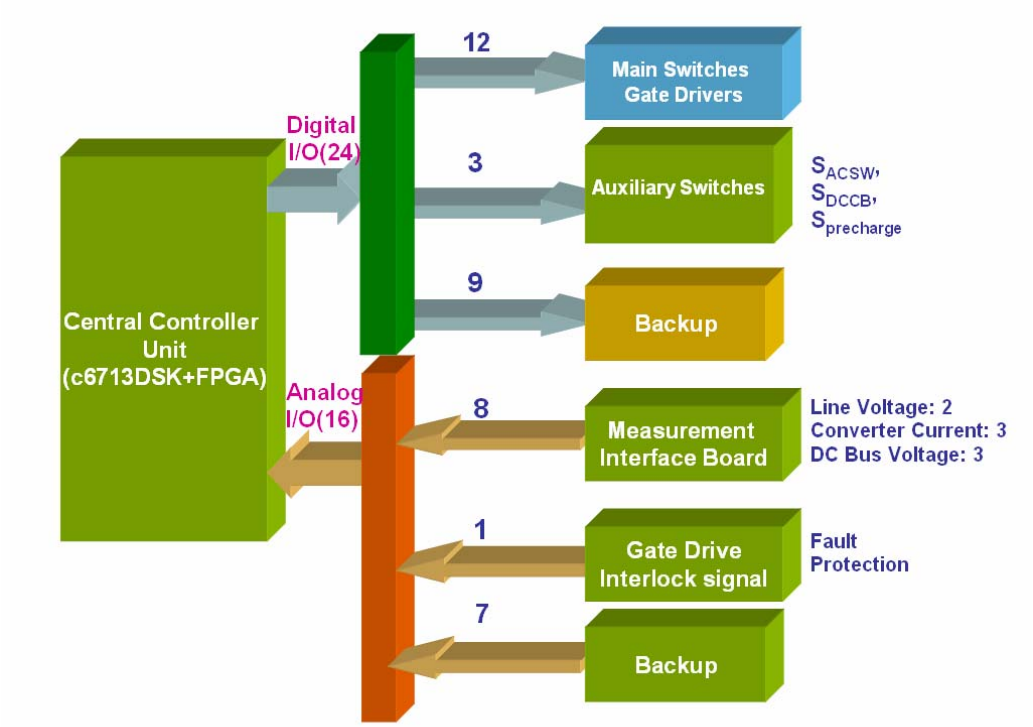


Figure 3.10. I/O configuration of the controller

The FPGA could be programmed directly with the JTAG cable. However, the program in FPGA will be cleared while the power is off. The flash memory XCV300 accompanying with the FPGA could also be programmed. It can be configured to load the program to FPGA automatically. Also, the program in the flash memory will not be cleared while the power is off.

3.3 Software Design

The software design of the control including three parts: DSP configuration setup, control algorithm implementation and Human Machine Interface design.

Firstly, the DSP need to be configured first before it starts to work properly. For example, without the PLL clock setup correctly we can't control the DSP's core frequency, peripheral clock frequency and EMIF frequency; without the memory allocation correctly the DSP are not able to access data inside FPGA. Just like "Windows" operating system in the

PC machine, DSP itself has its own real-time operating system, which is DSP/ BIOS. We need to configure DSP/ BIOS to make sure every part of the DSP cooperate each other correctly and efficiently.

Secondly, needless to say, the STATCOM control algorithm needs to be implemented.

Finally, to control the whole system we need a human machine interface. Traditionally, this system is analog system, which consists of push buttons, circuit breakers, and oscilloscope display. However the analog system is bulky and difficult to upgrade. In this controller design we will use LabView to build a Graphical based Human Machine Interface.

The data path of the whole system is shown in the figure below. It also denotes the DSP software routines.

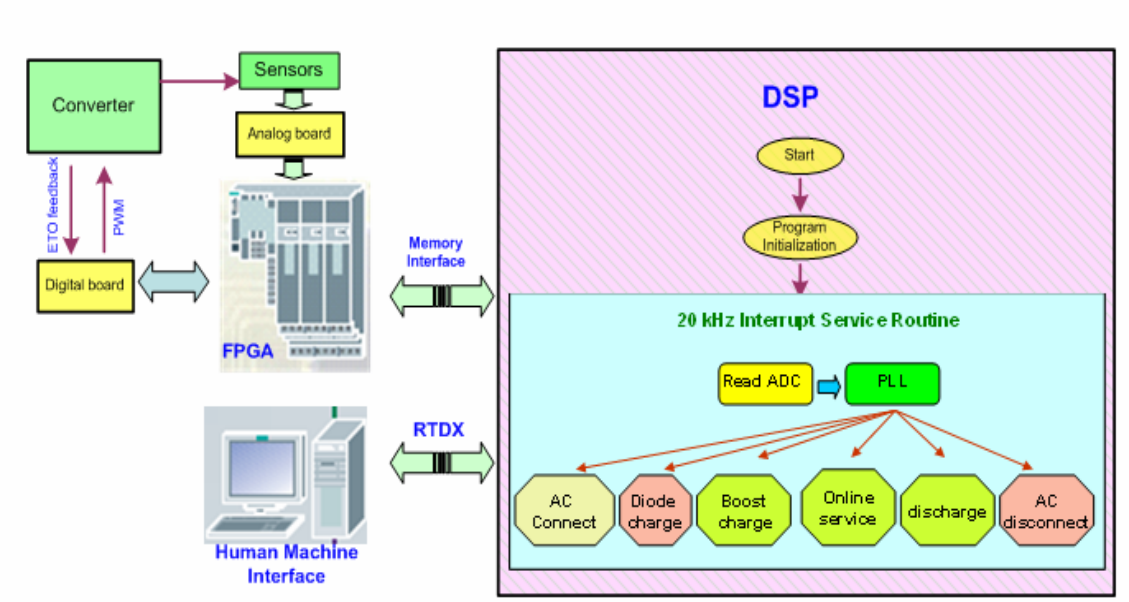


Figure 3.11. Data path and DSP software routines

3.3.1 DSP Configuration Setup

Figure 3.12 shows the architecture of TMS320c62x/C67x DSP. The C62x/C67x devices come with on-chip program and data memory, which may be configured as cache on some devices. Peripherals include a direct memory access (DMA) controller, power-down logic,

external memory interface (EMIF), serial ports, expansion bus or host port, and timers. Also there is PLL clock generator.

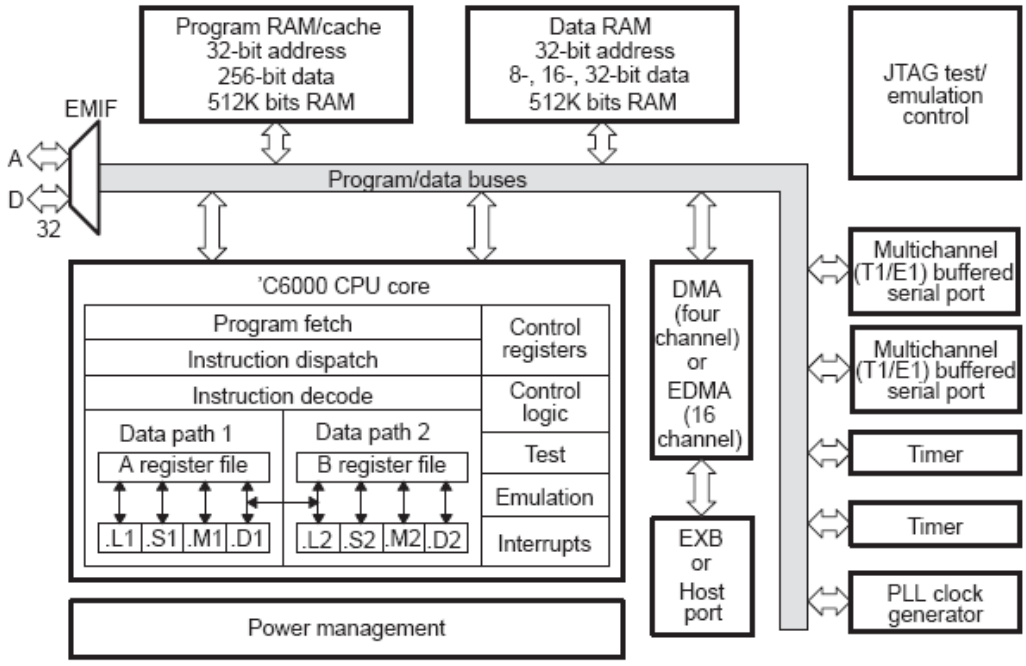


Figure 3.12. Architecture of TMS320c6713 DSP [18]

I. DSP Software-Programmable Phase-Locked Loop (PLL) Controller

The first thing to setup is the DSP's core frequency and EMIF frequency. The PLL controller (Figure 3.13) features software-configurable PLL multiplier controller, dividers (OSCDIV1, D0, D1, D2, and D3), and reset controller. The PLL controller offers flexibility and convenience by way of software-configurable multiplier and dividers to modify the input signal internally.

The resulting clock outputs are passed to the DSP core, peripherals, and other modules inside the C6000 DSP. From the block diagram we can choose internal clock OSCIN or external clock CLKIN as the base source of the PLL. Then this base clock will go through the Divider D0. It can be configure be divided by from 1 to 32. Then this will go through a multiplier, which can times the clock from 1 to 32. The clock signal out from the multiplier

goes through three different dividers D1, D2 and D3. The output of D1 is the DSP core frequency, D2 is peripheral frequency and the D3 is the EMIF frequency.

In this controller not only the DSP core frequency but also the EMIF frequency is very important. The AED-106 daughter board uses the EMIF clock as its base clock. Therefore every program in the FPGA depends on this EMIF clock. In our FPGA program design, the base clock is set to 50 MHz. Therefore we need to configure the EMIF frequency works at 50 MHz.

In our design we choose the on-board internal oscillator as the clock input. This oscillator works at 50 MHz. Then we configure the divider D0 to keep the frequency at 50MHz. The multiplier will times the frequency by 9. Therefore the PLL-out clock is 450 MHz. We want the DSP works at its highest frequency. Then the divider D1 divides the PLL out by 2. DSP works at 225 MHz. As we discussed above we need the EMIF works at 50 MHz. So the divider D2 divides the PLL out signal by 9. In c6713DSK support library, there is a source file names 6713DSK.C. It provide us an initialization function "DSK6713_init ()". Use this function we can setup the parameters for the DSP core frequency, EMIF and peripheral clock.

```
DSK6713_init () {
    ……
    /* Set main multiplier/divisor */
    PLL_RSET(PLLM, 9);                  // 50MHz  x 9 = 450MHz
    PLL_RSET(PLLDIV0, PLL_PLLDIV0_RMK(1, 0));  // 450MHz / 1 = 450MHz
    PLL_RSET(OSCDIV1, PLL_OSCDIV1_RMK(1, 4));  // 50MHz  / 5 = 10Mhz
    /* Set DSP clock */
    PLL_RSET(PLLDIV1, PLL_PLLDIV1_RMK(1, 1));  // 450MHz / 2 = 225MHz
    plldelay(20);
```

```
    PLL_RSET(PLLDIV2, PLL_PLLDIV2_RMK(1, 3));  // 450MHz / 4 = 112.5MHz
    plldelay(20);
    /* Set EMIF clock */
    PLL_RSET(PLLDIV3, PLL_PLLDIV3_RMK(1, 8));  // 450MHz / 9 = 50MHz
    ……
}
```

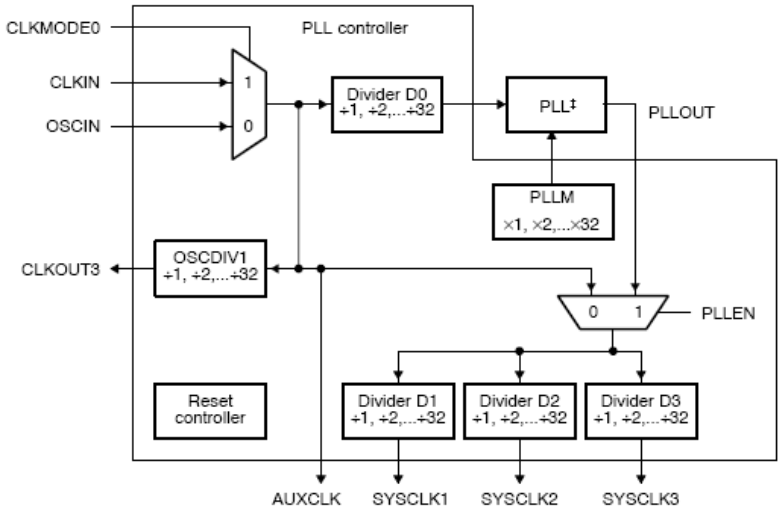


Figure 3.13. PLL Controller Block Diagram [19]

II. ***DSP Memory allocation***

As discussed before, memory allocation of the DSP is vitally important. Because we want to directly access the FPGA daughter card, we need to allocate a memory section for the FPGA. Figure 3.14 shows the memory map of 6713 DSK. The address within EMIF CE2 and CE3 are reserved for the daughter card. We will choose the address from 0xA0200000 0xA0300000 as our address section for daughter card. In the FPGA these address section is realized in the file selector.vhd module.

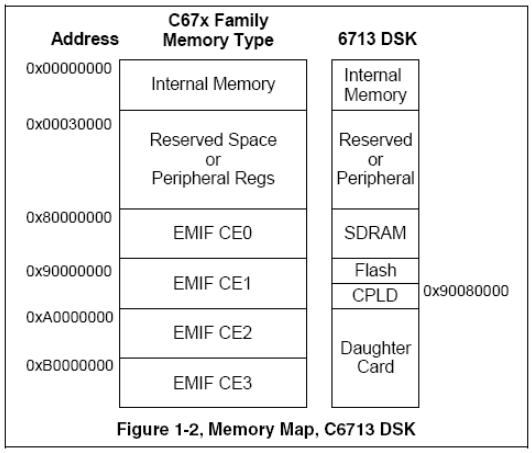


Figure 3.14. 6713 DSK Memory Map [16]

Because the FPGA daughter card has 16 channels A/D converter, 16 addresses are allocated for these A/D converter. Also to control the PWM generator module inside the FPGA four addresses are provided. Also we want to control the auxiliary switches also need to allocate one address. Please refer to the appendix file TVA_consts.h for the details of memory allocation.

III. ***DSP/ BIOS Configuration***

DSP/BIOS is a scalable real-time kernel. It is designed for applications that require real-time scheduling and synchronization, host-to-target communication, or real-time instrumentation. DSP/BIOS provides preemptive multi-threading, hardware abstraction, real-time analysis, and configuration tools.

The Configuration Tool has an interface similar to the Windows Explorer, and has multiple roles:

It lets us set a wide range of parameters used by the DSP/BIOS real-time library at run time.

It serves as a visual editor for creating run-time objects that are used by the target application's DSP/BIOS API calls. These objects include software interrupts, tasks, I/O streams, and event logs.

It provides visual editor as shown in Figure 3.15 to set properties for these objects

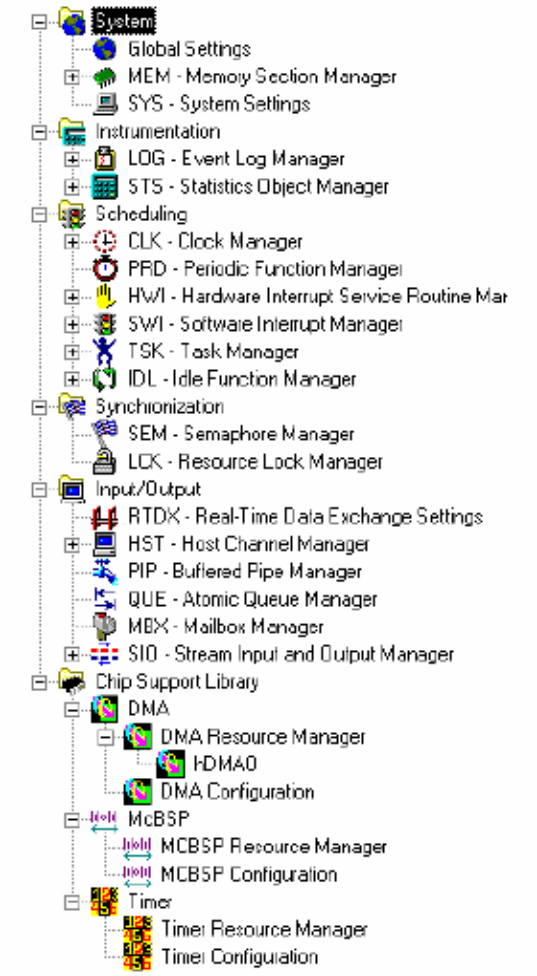


Figure 3.15. DSP/ BIOS configuration tool visual editor interface [18]

In this controller design I mainly use this visual editor to setup properties of the DSP. Because it's a graphical user interface and provides most of the control register address. Use this interface most of the DSP configuration can be easily done. Here just discuss the most important configuration parts.

Memory Section Manager (MEM). MEM allows me to setup the detail memory allocation. For example, what part of the programs is put inside Internal RAM, what is put in ROM or SDRAM. Also it allows specifying the heap size of the STACK.

Hardware Interrupt Service Routine Manager (HWI). The whole program is based on interrupt service routine (ISR). The timer0 will trigger this ISR in every 50 us (20 kHz). And ISR calls a function named applicationISR() to perform all the control algorithms.

Real Time Data Exchange Setting (RTDX). RTDX provides real time data detection during DSP operation. I will discuss in more details in the HMI chapter.

Timer. There are two timers in 6713 DSK timer0 and timer1. Here timer1 is used for the DSP/ BIOS as the reference clock. And timer0 is configure to trigger a HWI every 50 micro seconds. To setup the timer needs to create a timer setup files in the timer section of the configuration tools just like Fig. 3.16 shows. Then just provides the control parameters for the timer. For example, in the tab "Clock Control" we can configure the counter frequency of the timer0. Here we choose DSP core frequency divided by 4 as the timer counter frequency. That's 56.25 MHz. We want the timer will trigger an ISR in every 50 us. Then the period register value of the timer can be calculated:

$$period = 50 \times 10^{-6} \times 56.25 \times 10^{6} = 2812.5$$

Scaling this value to hexadecimal, its 0x00000AFC just like the value in figure 3.18 shows.

There are some other setups in DSP/ BIOS. Please refer to the appendix file RTDX.cdb for more details.

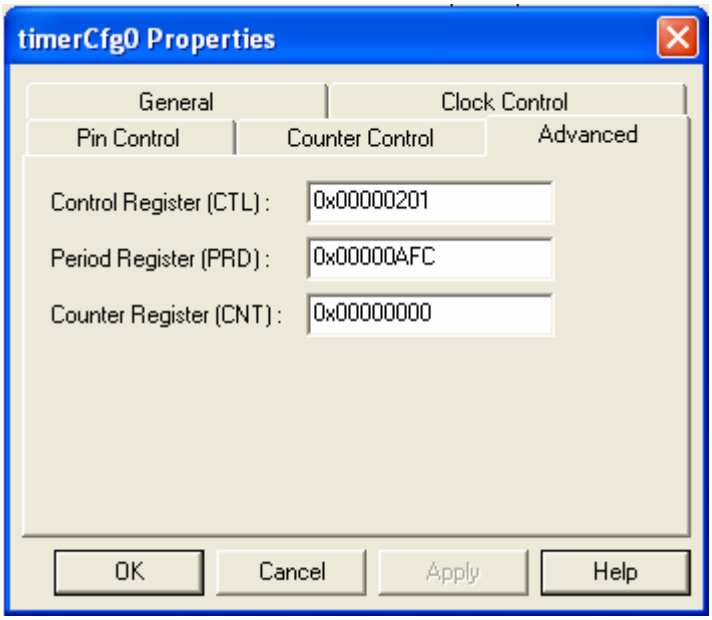


Figure 3.16. Timer0 configuration file

3.3.2 Control Algorithm Implementation in DSP Program

In figure 3.6, the diagram of the control block, there are PLL block, voltage and current compensators. Also to filter the noise we need to implement low pass filter. The voltage and current compensators basically are PI controllers. In this chapter we will talk about the implementation of these control blocks in DSP.

IV. ***PI Controller Implementation in DSP***

PI controllers are universally known because of their flexibility combined with the relatively easy tuning. This section describes the conversion from the continuous to the discrete time domain, which is essential for every implementation on a digital processor. A routine is then presented that implements the discrete time representation of the PI controller.

PI controllers are in most cases analyzed and tuned in the continuous time domain. The corresponding transfer function is given as:

$$U(s) = (K_p + \frac{K_i}{s})I(s) \qquad \text{Equation 3-1}$$

where I and U denote the input error signal and the controller's output. The typical Bode diagram of a PI controller is shown below:

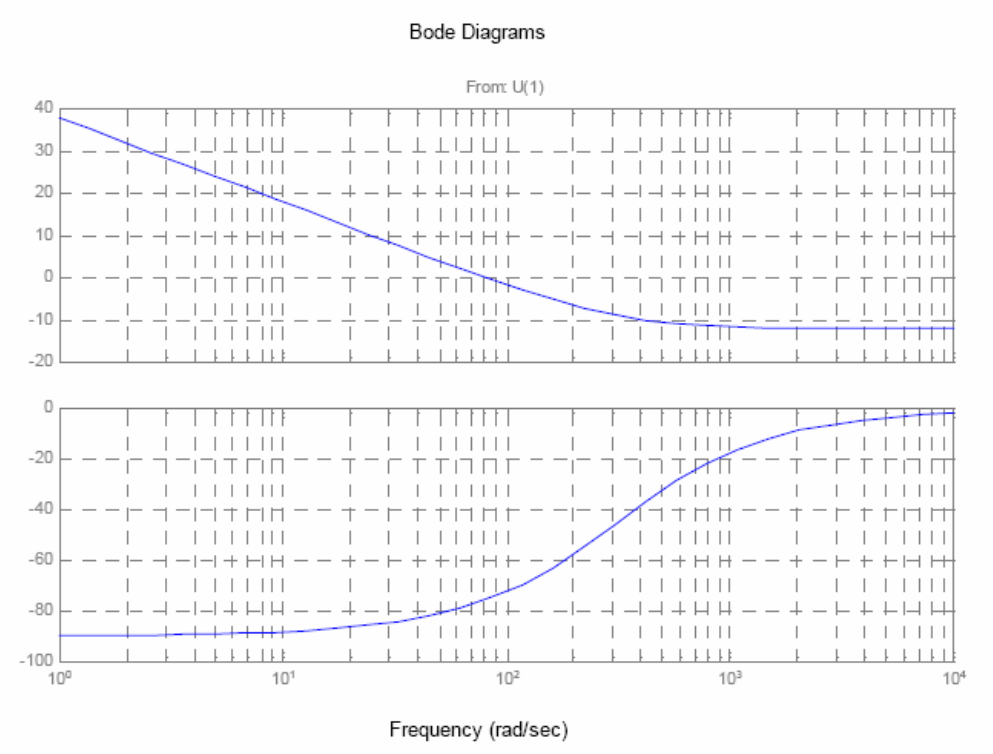


Figure 3.17. Typical bode plot of a PI controller

The transition from the continuous to the discrete time domain entails that the integral operation has to be approximated by a discrete summation. There are several methods for replacing the integral. Two of them will be discussed hereafter.

- Zero Order Hold (ZOH)

With this approach, the signal is sampled at the instant k and held constant until the next sampling instant k+1. The following figure illustrates this.

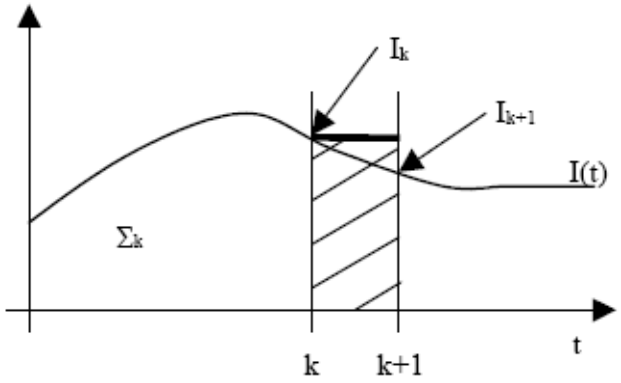


Figure 3.18. Integral with ZOH approximation

The integral operation is approximated by accumulating the rectangular areas. Denoting the sum at instant k with Σk, the signal at instant k with I_k and the sample time with T_{sample}, the "integration" is achieved by:

$$\sum_{k+1} = \sum_{k} + I_k \cdot T_{sample} \qquad \text{Equation 3-2}$$

The same equation can be expressed in the z-domain by:

$$z\sum(z) = \sum(z) + I(z) \cdot T_{sample} \qquad \text{Equation 3-3}$$

which leads to

$$\sum(z) = \frac{T_{sample} \cdot I(z)}{z-1} \qquad \text{Equation 3-4}$$

Therefore the continuous integration 1/s in equation 3-1 is replaced by $\frac{T_{sample}}{z-1}$, which leads to

$$U(z) = \frac{K_p z + K_i \cdot T_{sample} - K_p}{z-1} I(z) \qquad \text{Equation 3-5}$$

This corresponds to the following difference equation:

$$U_{k+1} = K_p \cdot I_{k+1} + (K_i \cdot T_{sample} - K_p) \cdot I_k + U_k \qquad \text{Equation 3-6}$$

- First Order Hold (FOH)

The Zero order hold method is simple and easy to implement. However it's not very accurate because during the sample period the input maybe not always the same. To improve the accuracy First Order Hold method is introduced below.

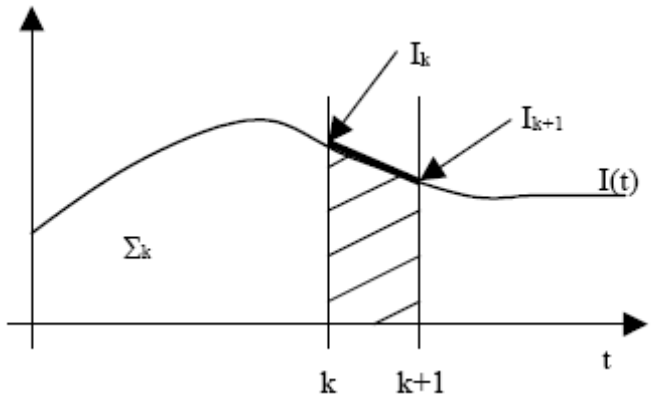


Figure 3.19. Integral with FOH approximation

The integral operation is approximated by accumulating the trapezoidal areas. Denoting the sum at instant k with Σ_k, the signal at instant k with I_k and the sample time with T_{sample}, the "integration" is achieved by:

$$\sum_{k+1} = \sum_{k} + \frac{I_k + I_{k+1}}{2} \cdot T_{sample} \qquad \text{Equation 3-7}$$

Following a similar procedure as in the previous section, this may be expressed in the z-domain by:

$$z\sum(z) = \sum(z) + \frac{T_{sample} \cdot I(z)(1+z)}{2} \qquad \text{Equation 3-8}$$

which results in

$$\sum(z) = \frac{T_{sample} \cdot (z+1)}{z-1} \cdot I(z) \qquad \text{Equation 3-9}$$

This time, the continuous integration 1/s in equation 3-1 is replaced by the whole factor preceding *I(z)*. The transfer function in the discrete domain is obtained from (3-1) as:

$$U(z) = \frac{(\frac{K_i \cdot T_{sample}}{2} + K_p) \cdot z + \frac{K_i \cdot T_{sample}}{2} - K_p}{z-1} I(z) \qquad \text{Equation 3-10}$$

This corresponds to the following difference equation:

$$U_{k+1} = (\frac{K_i \cdot T_{sample}}{2} + K_p) \cdot I_{k+1} + (\frac{K_i \cdot T_{sample}}{2} - K_p) \cdot I_k + U_k \qquad \text{Equation 3-11}$$

Based on equation 3-11, it's very simple to implement the PI controller in C language. Please refer to the appendix file TVA_control.c for more details.

V. ***Low Pass Filter***

To suppress the noise of the sensor signals, low pass filters are needed. The transfer function of a low pass filter is:

$$U(s) = \frac{1}{1+sT} \cdot I(s) \qquad \text{Equation 3-12}$$

where 1/T is the cut-off frequency of the low pass filter. Applying the FOH method, the equation 3-12 turns into:

$$zU = \frac{(z+1)I - (1 - f_{sample}T)U}{(1 + f_{sample}T)} \qquad \text{Equation 3-13}$$

which corresponds to the difference equation below

$$U_{k+1} = \frac{1}{1+f_{sample}T} \cdot I_{k+1} + \frac{1}{1+f_{sample}T} \cdot I_k - \frac{1-f_{sample}T}{1+f_{sample}T} \cdot U_k \qquad \text{Equation 3-14}$$

So to implement low pass filter just use equation 3-14. To change the cut-off frequency, T needs to be changed accordingly.

VI. ***Digital Phase Lock Loop Implementation***

Phase locked loops (PLL) with all ac/dc converters take an important role in providing a reference phase signal synchronized with the ac system. As we discussed above, to apply the

reactive power and active power decoupling control strategy, Park transformation is the key to transform ABC coordinate to DQ0 coordinate. And the Park transformation is based on this PLL signal. Therefore a stable and accurate phase lock loop is needed. In DSP we can implement a PLL loop in digital method.

The most common PLL topologies see (Fig. 3.20) [20] can be classified as zero-crossing structures in which the detection of phase and frequency disturbances is based on the zero crossing instants of the input signal, resulting in a slow structure. Additionally, the Voltage Controlled Oscillator (VCO) block demands a dc input signal. Hence, a Low-pass Filter is required, thus contributing to further constrain the dynamic performance of the PLL structure.

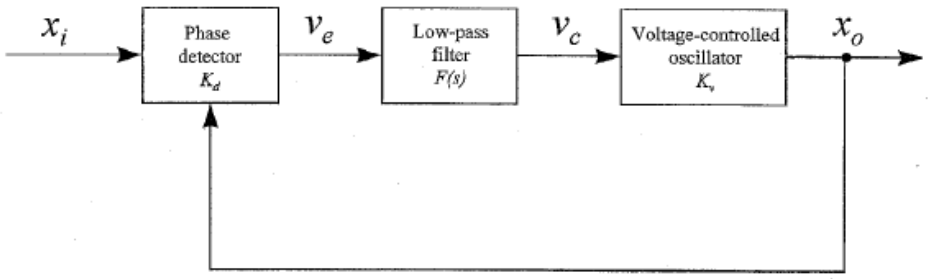


Figure 3.20. Basic topology of the PLL [20]

The three-phase PLL structure introduced in [21] is illustrated Fig. 3.21, where ω_{ff} is the feed forward frequency. This topology is based on the synchronism between the utility voltage vector and the synchronous reference frame. Setting the direct axis reference voltage (Vq) to zero and tuning the controllers to extinguish the error between Vq and Vq* result in the lock in of the PLL output on the phase of the utility voltage vector.

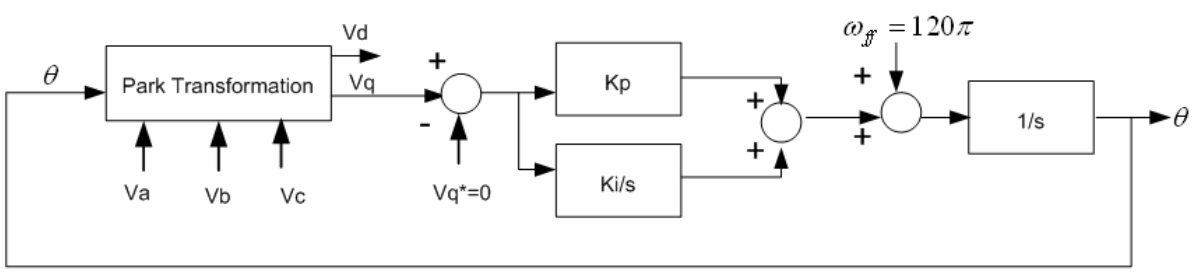


Figure 3.21. Three Phase PLL structure

Based on this structure, a DPLL block is created in the DSP program. Please refer to the appendix file TVA_PLL.c for more details. The experiment results for the DPLL are shown in figure 3.22. These figures are recorded through DSP. We can see that the DPLL gives accurate angle output of V_{AB} even there are harmonics in the voltage signals.

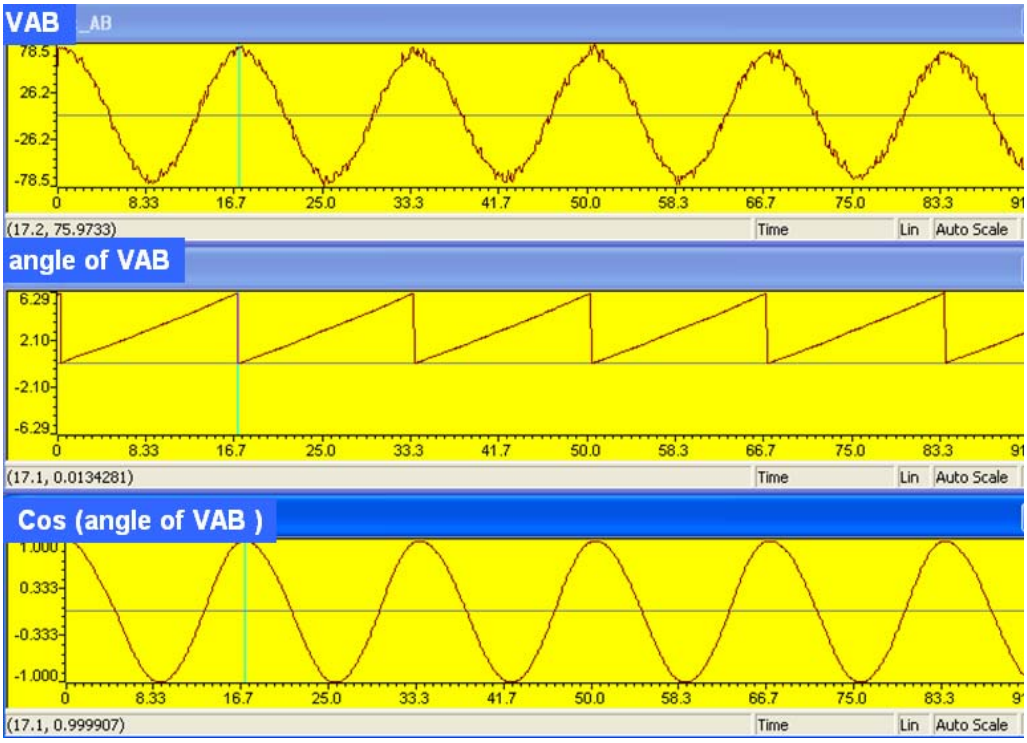


Figure 3.22. Experiment results of DPLL block in DSP

3.3.3 Human Machine Interface Design and Control Flow Chart

To fully control the STATCOM system, a Human Machine Interface (HMI) is needed. Figure 3.23 shows an analog HMI system for the STATCOM system [9]. This system

includes analog push buttons, analog voltage meters, current meters, circuit beakers, and oscilloscope. However the analog HMI have some limitations:

- Complicated and difficult to maintain. All the connections are electrical which are harder to maintain and design compare with software.
- Non-expandable. To add new functions, the circuits, the control panel, maybe even the cabinet need to be redesigned.
- Poor data display ability. Needs to use oscilloscopes to view the data waveform and various voltage/ current meters to monitor the data.
- Lack of signal processing ability.
- Bulky and Occupy a lot of space

Figure 3.23. Analog HMI for STATCOM Testbed System

Therefore the digital based Graphical User Interfaces (GUI) are becoming more and more popular now. A digital GUI should include the functions as the figure shows below. Comparing with the analog HMI, the digital HMI have a lot of advantages:

- Friendly graphical interface, easy to use and maintain.

- Almost unlimitedly expandable. All the display windows, all the push buttons and breakers are pieces of software and can be simply modified.
- Smaller space occupation.
- Huge data display and process ability.

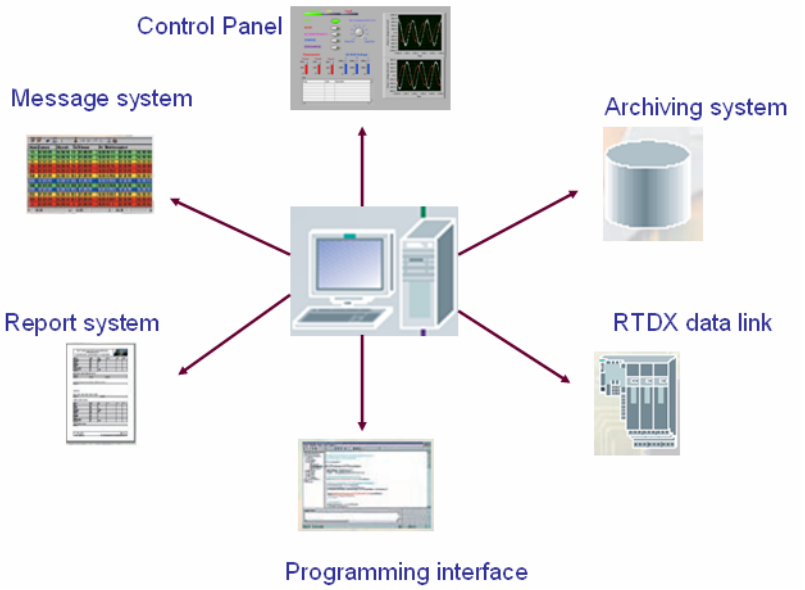


Figure 3.24. Functionalities of a SCADA/ HMI.

There are many ways to build this HMI. One is to use data acquisition cards (DAQ) to read the sensor signals and transmit them back to computer through PCI/ USB interface. This is the most straightforward ways and most of the digital HMI are base on this approach. However the DAQ are expensive and if there is a need to sense a lot of data, multiple DAQ are needed. This method therefore requires additional hardware.

Since all the sensor data are stored in the DSP memory, it is desirable to find a way to read DSP memory from a PC. In this way, no DAQ is required. Fortunately, Texas Instrument provides a possible solution to just achieve that. That is through the using of Real Time Data Exchange (RTDX) capability built in the TI DSP. With the help of RTDX there is no need to add additional hardware.

I. Real Time Data Exchange

TI's DSP provide the RTDX™ technology which provides industry's first continuous, real-time, visibility into the way target applications operate in the real world. RTDX permits developers to transmit and receive data between a host and a target DSP without stopping their applications and to view live or saved data via an easy-to-use Object Linking and Embedding (OLE) Application Program Interface (API). RTDX will transmit data at up to 8 kilobytes per second (bps), making it well-suited for our STATCOM control and data display. Figure below shows the communication route through RTDX.

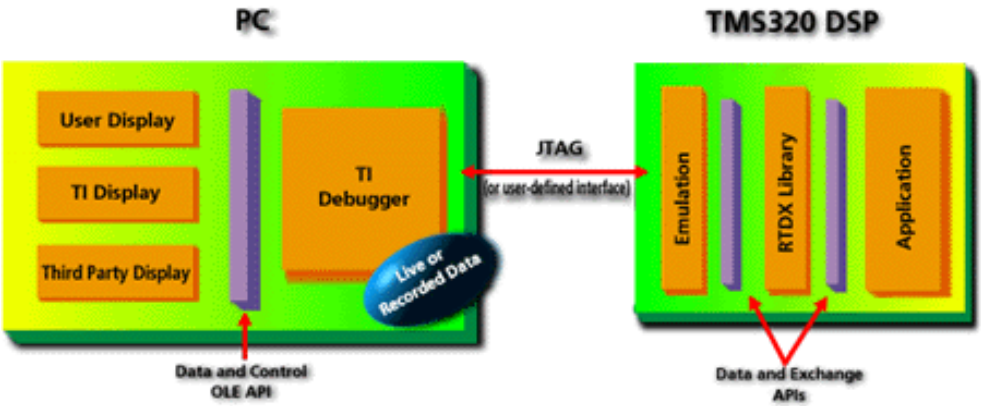


Figure 3.25. Communication through RTDX

RTDX consists of both target and host components. A small RTDX software library runs on the target application. The target application makes function calls to this library's Application Program Interface (API) in order to pass data to or from it. This library makes use of a scan-based emulator to move data to or from the host platform via a JTAG interface. Data transfer to the host occurs in real-time while the target application is running.

On the host platform, an RTDX Host library operates in conjunction with Code Composer™. Displays and analysis tools communicate with RTDX via an easy-to-use COM API to obtain the target data and/or to send data to the target application. We choose LabView as the host platform.

II. ***LabView Program***

LabView is very popular right now because it's nicely built virtue instruments. There are a lot of GUI Controls and displays inside the LabView library. Using them can dramatically shorten the program time and create beautiful interface. Labview also includes many digital signal process modules and data store systems.

The most important reason we choose LabView is because of the "Labview integration tool kit for TI's DSP". This tool kit includes a lot of modules which are nicely built for TI's DSP. Use this modules we can easily control the whole DSP operation. We can open Code Composer Studio™ by LabView; download coding to DSP; even use it to compile the codes. There are "RTDX_read" and "RTDX_write" functions in this tool kit. Use the first one to input data to DSP and use the latter to receive data from DSP. There are also DSP memory READ/WRITE functions. So by using this tool kit we can use RTDX communicate between PC and DSP.

However, there are two disadvantages for using LabView tool kit. Firstly, at the beginning the LabView tool kit needs to call CCS. All the control process need to use CCS. That means it's no way to decouple DSP and windows. Because CCS is a windows program if the PC died the CCS is also die. When this happens, the DSP will loose control. And even after the PC is restarted, we can not regain control of DSP. Another disadvantage is that the RTDX modules in LabView tool kit just like some black boxes. There is no clear description about the inside details of these modules. This limits the flexibility of the programmer.

The figure 3.26 is the control panel of HMI. To input control commands and monitor the operation of the STATCOM there are some push buttons and data display LED and data display windows.

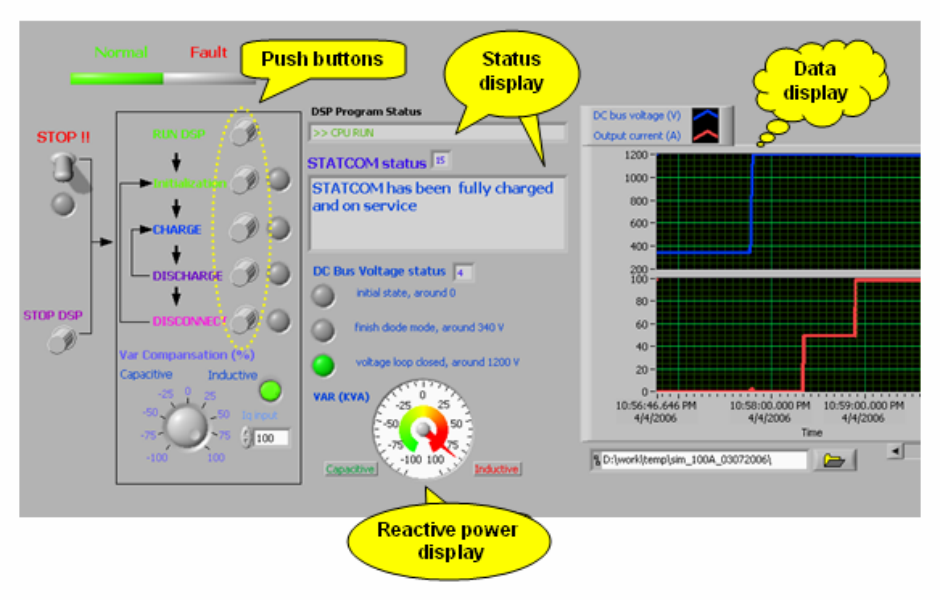


Figure 3.26. HMI control panel

The red “STOP” button is used for emergency stop. When it’s on, the DSP blocks all the gating signals of the VSC and reset the main AC switch and all other auxiliary switches. The STATCOM power stage is fully stop and disconnected from grid. But the STATCOM controller (DSP+FPGA) is still running. When fault is clear, the STATCOM is able to be reconnected to the grid and back to normal operation again. When this button is on, it blocks all other push buttons except the red “STOP DSP” button. This interlock is used to prevent inputting wrong commands during fault situation.

The “STOP DSP” button is used to stop the operation of the DSP. This is used to shut down the whole system including the STATCOM power stage and the controller units.

There are other five push buttons, which are “RUN DSP”, “Initialization”, “Charge”, “Discharge”, “AC disconnect”. These five buttons control the STATCOM operation sequence. For safety issues there is some control logic for these buttons. The buttons can’t be trigger only when the conditions are satisfied. I use the “DC bus voltage status” as the conditions. If the DC bus voltages are not in the specified range the button won’t be triggered.

Figure 3.27 shows the control logic of the push buttons. The operation flow chart is also shown in this figure.

The knob beneath the push buttons is used to control the output reactive percentage. The maximum output reactive current is 100A in this experiment.

There are also DSP status display, DC bus voltage status display, STATCOM status display and output reactive power display in the panel. By them we can monitor the status of the whole system.

The big data display window can display various data. For example, DC bus voltage, output current RMS value. Limited by the small data rate of the RTDX (only 8 kbps), the HMI can only display data which frequency is slower than 30 Hz.

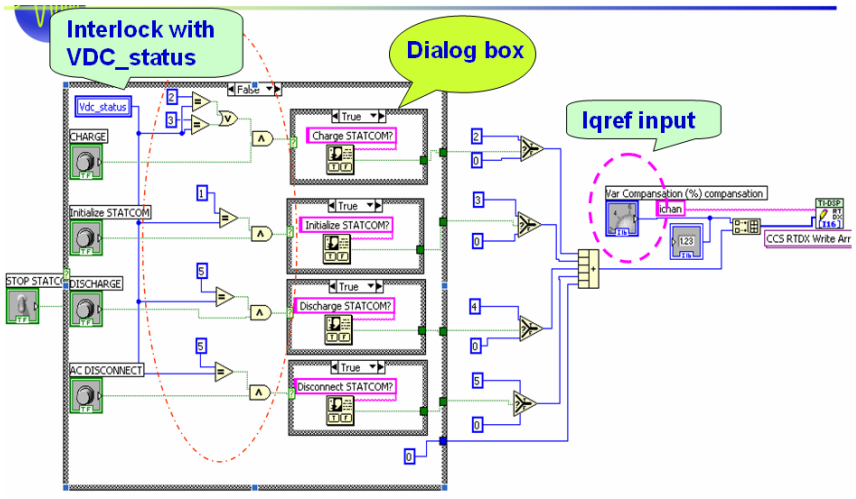


Figure 3.27. Interlock for push buttons

Figure 3.28 shows the STATCOM operation flow chart. There are six operation modes: Initialization, AC connect, charge, online service, discharge and Ac disconnect.

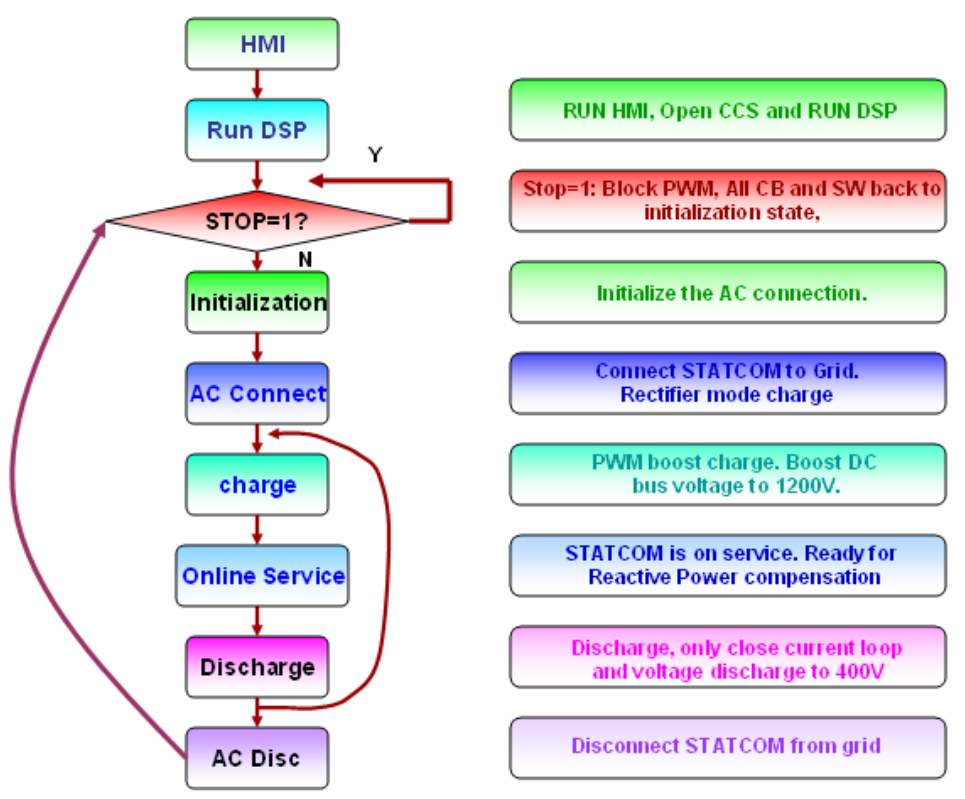


Figure 3.28. STATCOM operation flow chart

A "While loop" in the LabView program will read/write the RTDX channel periodically. The frequency can be modified. However because the data rate is 8 kbps, there is no much flexibility in the frequency selection. Now 33Hz is selected for the data collection. Another thing needs to mention here is that the RTDX channels creation. At the beginning I created different channels for different variables, five channels for outputs and 2 channels for inputs. However this will cause the DSP to call RTDX function too many times. The RTDX is a Hardware Interrupt. And it's reserve HWI and has almost the highest priority among DSP's HWI. Though the RTDX runs very fast, if call HWI too many times it will still cause some delay problems. So finally I only use two channels for the input/output. Use two arrays to group the input and output variables. Then DSP only needs to call RTDX twice to get all the information.

Also to speed up RTDX running, the RTDX code and DSP/BIOS code need to be put in IRAM instead of SDRAM. Because SDRAM runs much slower than IRAM. This will dramatically increase the time cost of the program.

3.4 Hardware-in-the-loop Real-time Simulation

The AC source at NCSU's SPEC lab can provide 480V 500A capability. For scale down experiment we choose 100A as the maximum output current. The parameters of the experimental system are shown in the table 3-1.

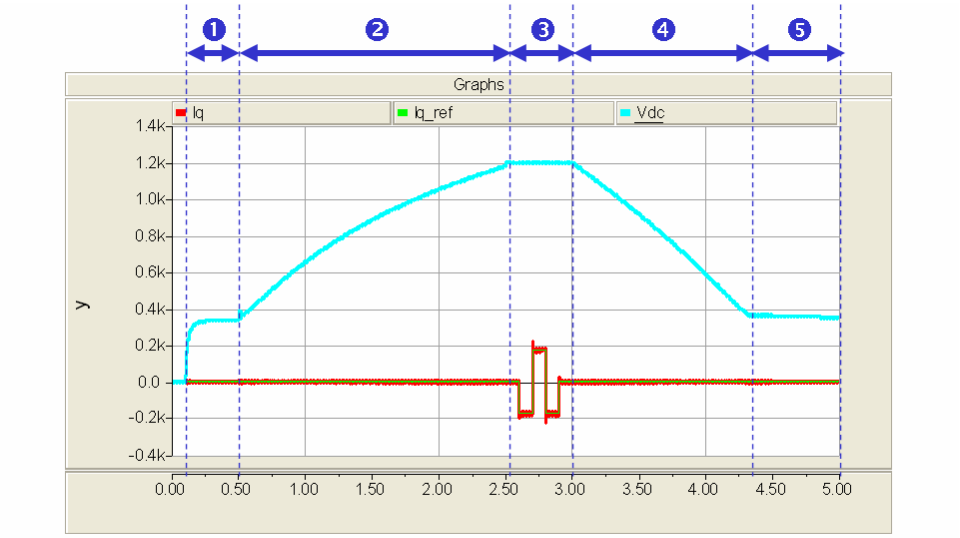
Table 3-1. Experimental system parameters for the 4.5 MVA CMC-based STATCOM

Three-Level ETO-based CMC STATCOM	3-Phase 3-Wire Wye-Connection
Configuration	Balanced
Individual DC Bus Voltage	1200±10%
Rated RMS Reactive Current	100A
Capacitor Impedance	(0.5m-j/(ω ×1.7mF)) Ω/Phase
Individual Switching Frequency	1kHz
Coupling Reactor Impedance	(31.5m+jω ×2.8 m H) Ω/Phase
PCC Line-to-line Voltage	480V

Based on the small signal model, external loop compensator H_E and current loop compensator H_{id}, H_{iq} have been designed to achieve fast and robust response. The bandwidth of the external voltage loop and current loop are 20 Hz and 200 Hz respectively. Off-line simulation has been done in PSCAD and the results are showed in figure 3.29 [15]. There are five operation modes of the STATCOM:

- Pre-charge mode. STATCOM operates like diode rectifier and will be charge to 340V.
- Energization mode. The DC bus voltages will be charged to the set point 1200V. In this operation mode we only need active power to charge the capacitor. So only current loop is closed. The charge speed depends on the d-channel current reference I_{dref}. I_{dref} is negative value. I_{qref} is zero because right now no reactive power output.

- Online service mode. After DC bus voltages are charged to the set point. Close external voltage loop to maintain voltage stable in 1200V. Now STATCOM is ready to output reactive power. The output reactive power will follow the q-channel current reference I_{qref}.
- De-Energization (discharge) mode. Like energize mode. Only need to output active power. Then only close current loop and set I_{dref} to positive value to discharge the energy. I_{qref} is zero.
- Post-discharge mode. The STATCOM is disconnected from grid and is discharged by the capacitor parallel resistor.



❶ pre-charge, ❷ energization, ❸ online service ❹ de-energization, ❺ post-discharge

Figure 3.29. Off-line simulation for the 480V 100A experiment [15]

The off-line simulation in PSCAD is in s-domain and it assumes the system is continuous. It can verify the control design and STATCOM operation procedures. However, the real controller is in z-domain because all the data process in DSP and FPGA are discrete sampling data. Therefore the off-line simulation is not good enough to simulate the whole system performance. Further more, the PSCAD off-line simulation is only software

simulation and it can't be used to verify the hardware setup, for example the sensors, circuit breakers, and hardware protection circuit and the ETO devices.

In order to validate all the hardware works properly, a hardware-in-the-loop real-time simulation was also carried out. Different with the off-line simulation in software, the on-line simulation was simulated inside the DSP hardware and all the hardware has been tested. The only difference between the real-time on-line simulation and the real STATCOM operation is that there is no power on the power stage and the sensor feedback signals are only used as protection instead of feedback of the control loop.

Because the power stage has no power we need to simulate the "feedback" of the power stage during different operation modes. The best way is to create a STATCOM power stage model inside the DSP software which can simulate the transient response of the power stage. According to the average model of CMC STATCOM in equation 2-6 and 2-7, an on-line STATCOM power stage average model is created within DSP program. Compare with the off-line simulation results the on-line simulation are more accurate because all STATCOM hardware is operating in real time exactly like the real experiment except without AC feeder power. The DSP code for the STATCOM average model is shown below:

```
#include "tva_types.h"
void statcom(const TVA_DQO_DATA duty, const float delta_t, STATCOM_PARAM* prSTATCOM){
    const float c=1.7e-3;                   // DC capacitance
    const float L=2.8e-3;                   // Coulping inductance
    const float R=31.5e-3;                  // Coulping resistance(ESR of inductor)
    const float vsd=480.0;                  // D-channel grid voltage (1.22*phase_va_pk)
    const float omega=376.9911;// 2*pi*60
    float delta_v;                          // voltage difference
    float delta_id;                         // D-channel current
    float delta_iq; // Q-channel current
    float delta_vtemp1 = duty.D*(prSTATCOM->id);
    float delta_vtemp2 = duty.Q*(prSTATCOM->iq);
    // capacitor voltage variance
```

```
    delta_v=(delta_vtemp1+delta_vtemp2)*delta_t/(3*c)*(-1);
    // D-channel current variance
    delta_id=((duty.D*(prSTATCOM->e)-vsd)/L+omega*(prSTATCOM->iq)-
    (R/L)*(prSTATCOM->id))*delta_t;
    // Q-chanel current variance
    delta_iq=(duty.Q*(prSTATCOM->e)/L-omega*(prSTATCOM->id)-
    (R/L)*(prSTATCOM->iq))*delta_t;
    prSTATCOM->e=(prSTATCOM->e)+delta_v;
    prSTATCOM->id=(prSTATCOM->id)+delta_id;
    prSTATCOM->iq=(prSTATCOM->iq)+delta_iq;
}
```

In this STATCOM model, capacitor and inductor parameters are implemented. *prSTATCOM.E* simulates the average DC voltage. *prSTATCOM.id* and *prSTATCOM.iq* simulate the D-channel current and Q-channel current respectively.

Fig. 3.30 shows the detail block diagram of the hardware-in-the-loop real-time simulation. All the hardware, include the DSP, FPGA, power stage and measurement system are working exactly as the reality condition, except the power stage is not connected with the power source. The output of the feedback loop d_d and d_q is feeding into the DSP STATCOM average model. The average model then generates the STATCOM DC bus average voltage E, D channel current i_d and Q channel current i_q. These signals are used as the feedback signals of the control loop. Then the close loop control design can be verified inside DSP. The generated duty cycle commands are sent to the SPWM generator inside FPGA and generate the gating signals. Then the gating signals are transferred to optical signals by the digital board and sent to the power stages. Therefore all the hardware is tested in the hardware-in-the-loop simulation. Table 3-2 shows the comparison between off-line software simulation and the hardware-in-the-loop simulation.

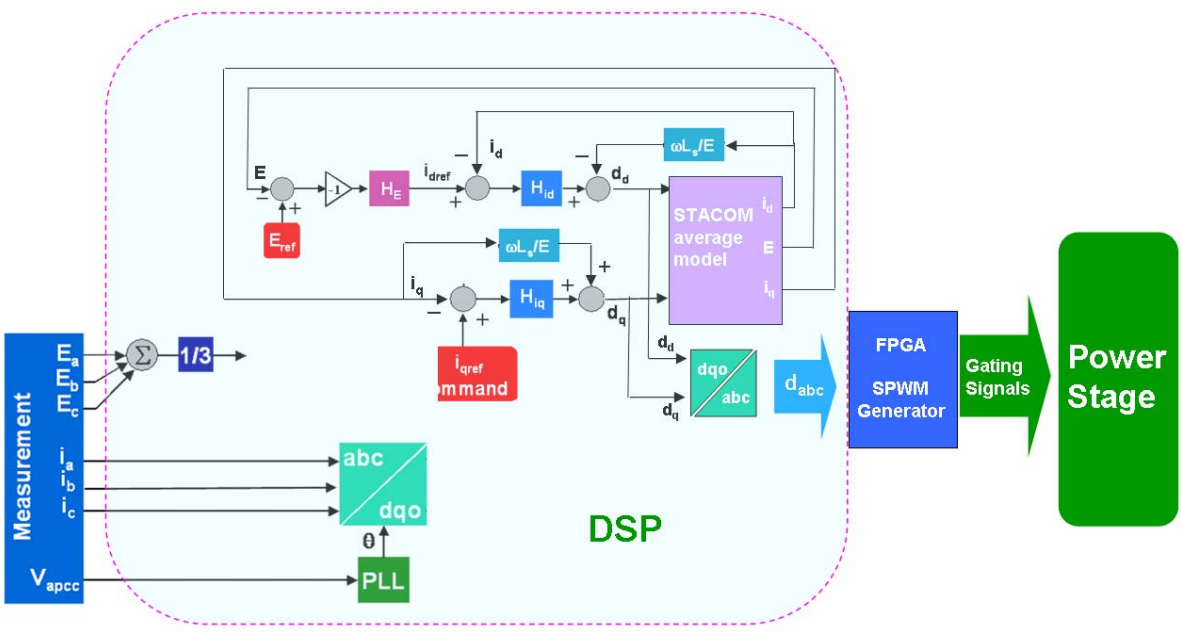


Figure 3.30. Hardware-in-the-loop real-time simulation block diagram

Table 3-2. Comparison between offline software simulation and hardware-in-the-loop simulation

	Off-line simulation	Hardware-in-the-loop real-time verification
Control design and operation modes	Yes	Yes
DSP hardware & software	No	Yes
FPGA hardware & software	No	Yes
Sensors	No	Yes
Auxiliary switches	No	Yes
ETO devices	No	Yes
Hardware connections	No	Yes

The simulation results are captured in DSP Code Composer Studio™. Figure 3.31 shows the DSP online simulation results. The first graph is DC voltage E; the second graph is the d-axis current I_d; the third graph is the q-axis current I_q. From 0ms to 708 ms, the STATCOM is in boost charge mode. Idref is set to -10A. From 708 ms to 1296 ms DC voltage loop is closed and DC voltage maintains 1200 V. From 916 ms to 1106 ms I_{qref} is set to -173 A. STATCOM works in capacitive mode. We can see that I_q follows the reference very well and

I_d has some transient response. From 1106 ms to 1296 ms, the I_{qref} is set to 173 A, STATCOM works at inductive mode. After 1296 ms, the STATCOM operates at discharge mode, I_{dref} is set to 10A I_{qref} is 0. The results of the on-line simulation verify the compensator control parameters design in z-domain and the transient responses are good.

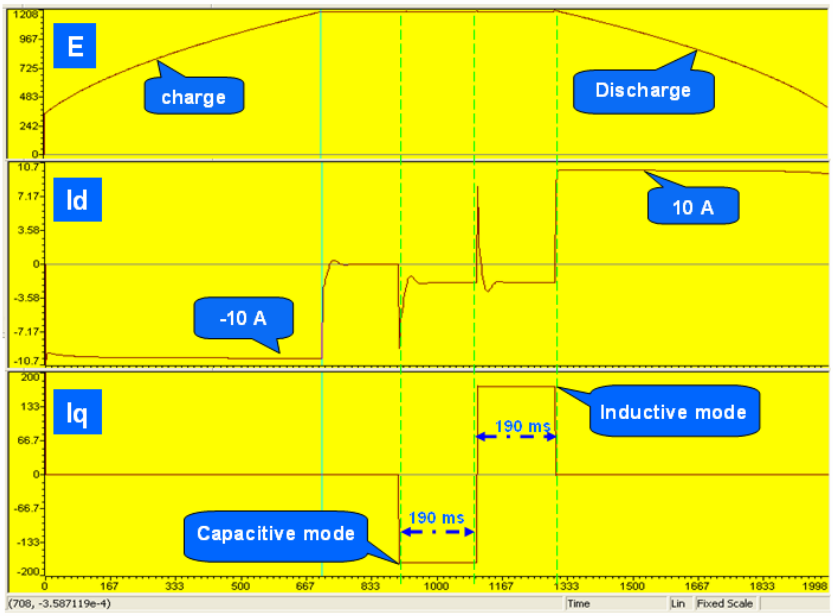


Figure 3.31. Hardware-in-the-loop real-time simulation results

Again, all the hardware and operation procedures are verified in the hardware-in-the-loop simulation. For example, during the energization operation mode, you can hear the sound of the pre-charge circuit breaker is on when it shorts the pre-charge resistors. Also when there are switching commands sent to the devices you can hear the ETO devices switching. As mentioned before, the digital board can compare the feedback signals of ETO with the switching signals, if there is any difference it will shut down the STATCOM and generate fault signals to the DSP. This fault signals then will also be displayed on the screen of the HMI. All these operations can be simulated during the hardware-in-the-loop real-time on-line simulations.

Finally, the off-line simulations and the DSP based real-time on-line simulations verify all the software and hardware design. The STATCOM controller is ready for actual power test. Figure 3.32 shows the final experiment setup.

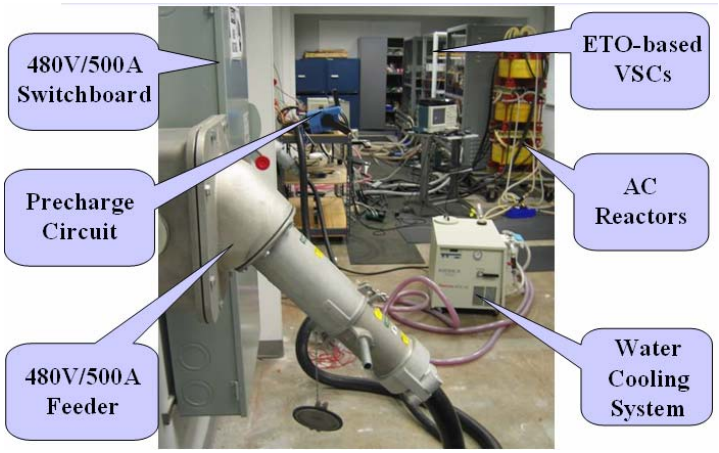
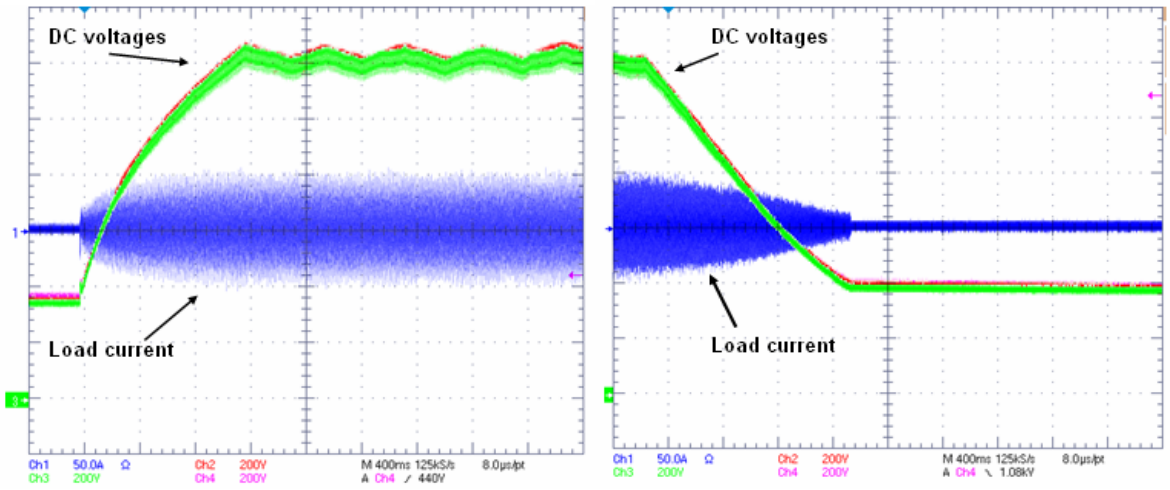


Figure 3.32. STATCOM experiment system setup

3.5 Experiment Results

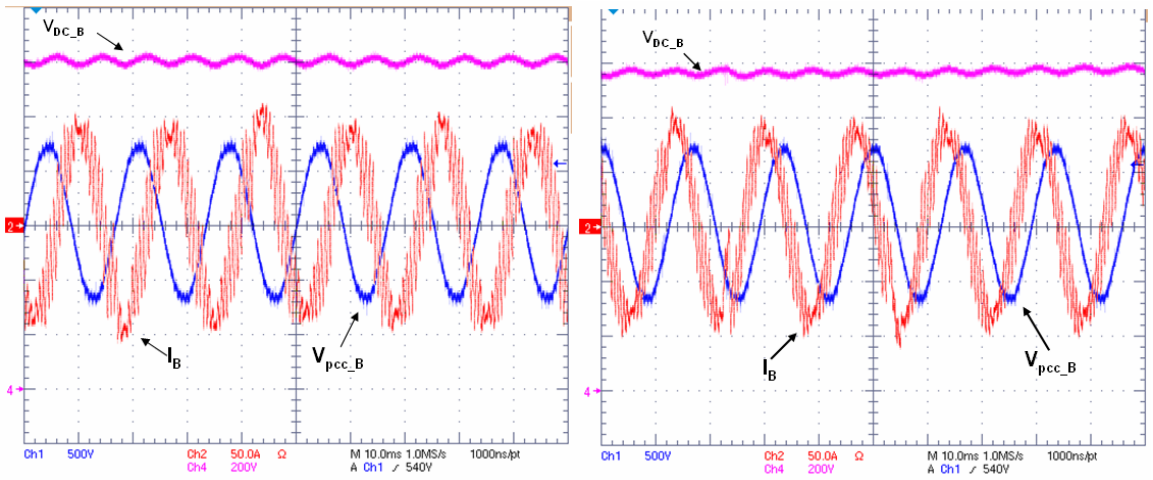
Figure 3.31 (a) shows the STACOM energization process. After the STATCOM finishes the diode rectifier mode the DC voltages are stable around 340V. At this time, give a negative constant I_{dref} can charge the capacitor at constant speed. Here the I_{dref} is equal to minus10A and the I_{qref} is 0A. When the three phase DC voltages are equal to 1200V, voltage loop is closed to maintain the DC voltages at this point. Increase or decrease the I_{dref} amplitude will shorten or increase the charging time. Figure 3.33 (b) shows the STATCOM de-energization process. When DSP receives the "Discharge" command, the STATCOM enters "de-energization" mode. Similar with the energization process, also gives a constant I_{dref} but at this time it's a positive value. Here I_{dref} is equal to 10A. We can see that the DC voltages are discharged linearly to around 400V.



(a) (b)

Figure 3.33. (a) STATCOM energization; (b) STATCOM de-energization

Figure 3.34 (a) shows the typical experimental results during steady-state 100A capacitive current output (I_{qref} is equal to -100A). Here, the current direction is defined to flow from converter out to the grid. During capacitive mode STATCOM output reactive power to grid. Phase current lags phase voltage 90 degree. Fig. 3.34 (b) shows 100A inductive current output. Inductive mode STATCOM absorbs reactive power and current leads voltage 90 degree.



(a) (b)

Figure 3.34. (a) 100A capacitive mode (b) 100A inductive mode

Figure 3.35 shows the dynamic response of the STATCOM system. Fig. 3.35 (a) shows the STATCOM changes from stand-by mode to 100A capacitive mode. During stand-by mode, STATCOM only maintain DC bus voltage at 1200V and input 0 reactive power to the grid. Therefore at stand-by mode the current is in phase of phase voltage. Fig. 3.35 (b) shows the dynamic transient from 100A capacitive mode to 100A inductive mode. Fig. 3.35 (c) shows the inverse transient: I_{qref} changes from 100A to -100A.

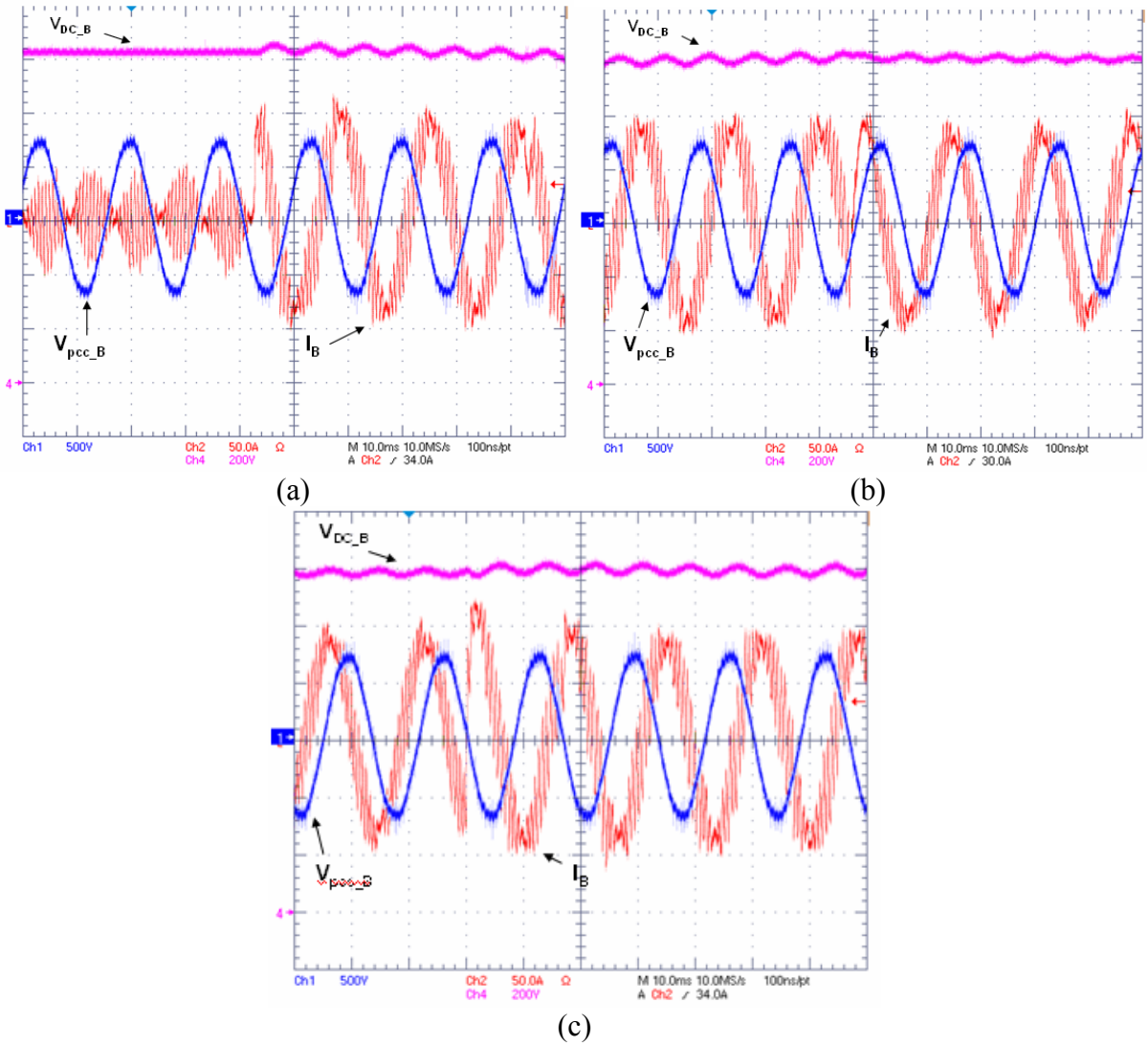
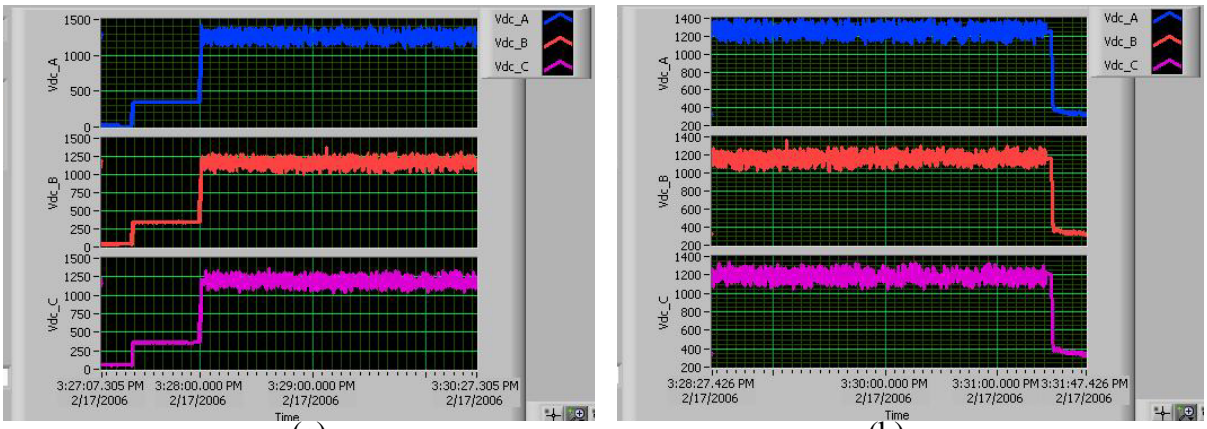


Figure 3.35. (a) Stand by mode to capacitive mode (b) capacitive mode to inductive mode (c) inductive mode to capacitive mode



(a) (b)

Figure 3.36. Waveforms capture in HMI. (a) Charge mode to Online Service mode, (b) Online service mode to Discharge mode

3.6 Conclusion

In this chapter, control strategy of CMC-based STATCOM is derived based on the work of CMC-based STATCOM modeling and control.

The details of the STATCOM controller hardware design has been discussed, including why to choose the TI's TMS320c6713 DSP as central controller and why the DSP + FPGA architecture is necessary.

The operation system of DSP is the key of the controller design. The configuration of the DSP has been introduced in the third section of this chapter, including the DSP software programmable PLL set up and DSP memory allocation. This part is very important, because before the DSP running any program, it needs to be properly configured first. DSP/ BIOS is the basic operation system of the DSP. It is the brain of the DSP and controls all the hardware resource of it. Though all the function can be programmed only use the C script, it's very tedious and easy to make mistakes. DSP/ BIOS configuration tool provide us a graphical interface to access all the function of the DSP. By the configuration tool, we can setup memory manager to configure the memory allocation for different program sections. We can create a hardware interrupt service routine for the control function. And this ISR is

triggered by timer0 at 20 kHz. The timers set up also can be finished in DSP/ BIOS configuration tool.

The next section of this chapter talks about the control software implementation. Firstly, PI controller, low pass filter and digital PLL block are introduced and the method of implementation in DSP has been discussed in detail.

To fully control the STATCOM a digital Human Machine Interface is introduced. Real time data exchange technology provides us a new way to do the communication between DSP and PC. By this new method we can use LabView to build a graphical user interface for the STATCOM without using a DAQ.

A three-level CMC-based STATCOM demo system is introduced. To verify the control scheme, hardware-in-the-loop simulation has been conducted. In this simulation the whole system works as the real operation situation except the power stage is not energized. The simulation tested the compensator parameters and the whole system hardware, software design and STATCOM operation procedure.

Finally, 480V 100A experiment has been conducted and the results have been published. The experimental results show not only the stable and fast response of the STATCOM, but also low harmonics, excellent DC voltage control and cooling performance.

In summary, through such an experiment, STATCOM system with high performance and low cost is illustrated and tested.

Chapter 4 Modular Digital Controller for CMC-based STATCOM System

Conventionally, in most of the high power applications, the centralized topology controller is widely used. In this control architecture, the central controller performs all the control functions, receives all the sensor signals, and generates the modulation commands. After that, the gating signals are generated for each device in the converters. The control function is quite complex. If need to increase the power rating, which means the number of the modular converters increases, then the load of the controller will increase dramatically.

For the above architecture, between the controller and the power converters, there are large numbers of connections. Most of them are electrical wires for analog sensing signals. However for the high power application, EMI noise is an issue. The sensing signals which are transmitted in the long electrical cables are disturbed easily. To eliminate this noise, digital filters are needed in the control software. This also increases the load of the controller and may jeopardize the control stability because the filter delays. Therefore, a very powerful central controller is needed for the future update in this topology.

Also the interfaces of the controller need to be update if more modular converters are added, because all the control signals are from the central controller units. Although the converter has already been modular in this example, the system still could not achieve modularity because of the complex connection interface between the controller and the power stage.

The centralized controller structure has the following main drawbacks:

- Poor system expansion capability
- Large number of wire connections between the power stage and the controller

- Poor noise susceptibility
- Low system reliability as each wire is a failure node

Here let's introduce a whole new topology of the controller: modular controller topology. The modular controller's structure is Central controller + local controllers. Local controllers are designed for each modular converter. It collects the sensor signals and sends them to the central controller by series communication protocol. The central controller then performs the control function and sends out the switching states to local controller. The local controller decodes the switching states to detail gating signals and directly controls the module converter. If any fault happens, the local controller is smart enough to shut down and protect the module converter and informs the central controller what happens.The local controller not only provides a "brain" for the module converter but also take care of some work which is supposed to be taken by the central controller.

The following features are considered for the defined modular controller

- Integrated current, voltage and temperature measurement.
- Isolated gate drive for both top and bottom switches of the phase leg.
- Series communication between local and central converter through optical fiber.
- Protection functions including over current, over voltage, and over temperature protection

4.1 Selected Modular Controller Topology.

There are different network topologies for example Bus topology, Ring structure, Star topology, and Tree topology. In [22], all of these structures are compared and an optimized Star topology is proposed, which is shown in figure 4.1.

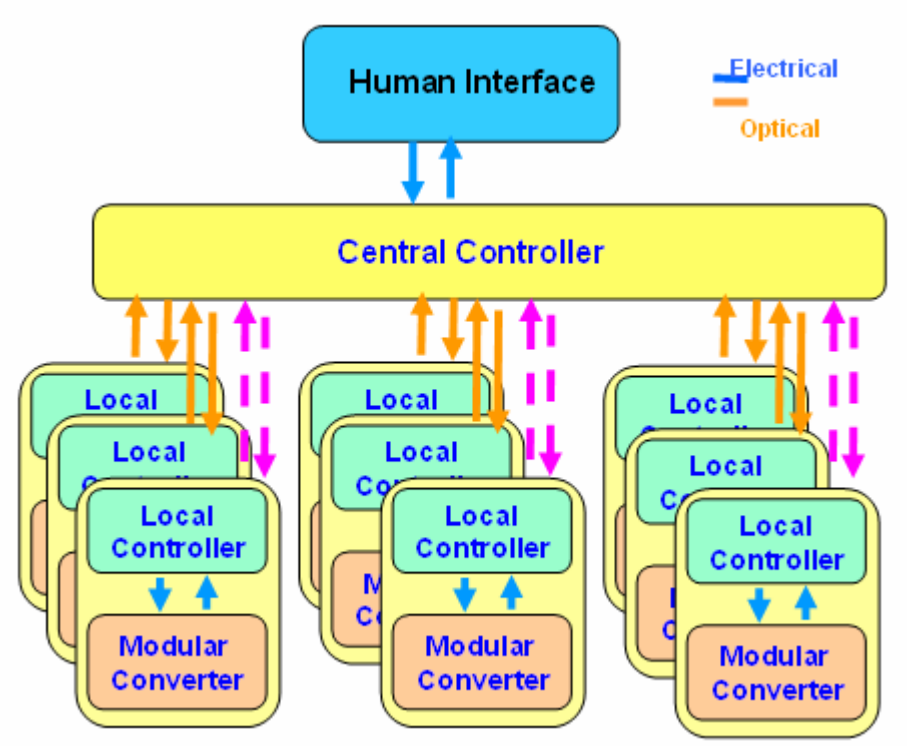


Figure 4.1. Optimized STAR network for 7-level CMC-based STATCOM [22]

The central controller is a digital controller that gathers all the required system information and performs the control operations to generate control commands. The local controller will generate the switch signals to control the switches in the modular converter.

The communication between the central controller and the local controller can be implemented with simple asynchronous serial communication. Each local controller shares the same command from the central controller or gets data according to the address information and then generates various switching patterns for the modular converters. For the data return path, each local controller has an optical link with the central controller. This optical fiber will include all the voltage, current, temperature and fault information for each modular converter. They are sending back according to the predefined sequence.

The proposed modular controller architecture has the following advanced features:

- Improved reliability & isolation. In the proposed architecture, the central controller is communicating with the local controller via optical links. The connection numbers are greatly reduced and there is no long analog wire. With optical fiber link, the transmitted data is less susceptible to noise. In addition, optical link provides great isolation between the converter and the central controller. Thus the system reliability is highly improved.
- Improved Expansion Flexibility. With the local controller, the system is truly modularized. Each module converter has one local controller as its "brain". To increase the system power rating, just simply add more module converters and local controllers. Since the communication protocol and control parameters are the same, there is no need to modify local controller. Just add more optical fibers and modify part of the control software in the central controller.

4.2 Digital Modular Controller for 7-level CMC-based STATCOM

Figure 4.2 shows the detail structure of the modular controller for a 7-level CMC-based STATCOM. The red lines are electrical signal data path, and the blue lines represent the optical data link. The interface boards R1 and R2 are used to transfer electrical signals to optical signals. Interface boards T1 and T2 are used to transfer optical signals to electrical signals.

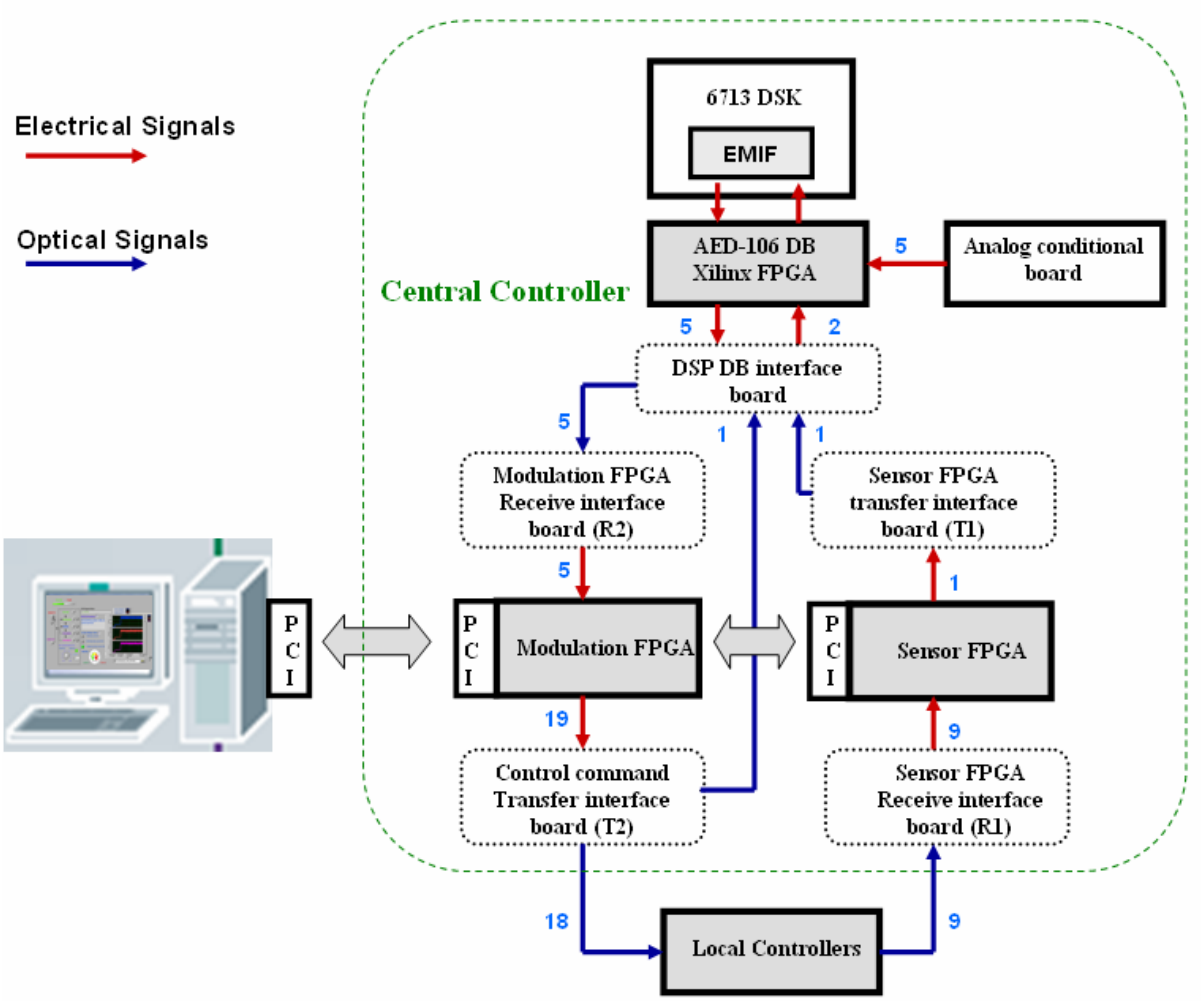


Figure 4.2. Modular controller detail structure for 7-level CMC-based STATCOM

A 7-level CMC-based STATCOM is a very complex system. There are nine module converters (three converters at one phase). Then we need nine local controllers. It's a huge system we need to process nine DC bus voltages signals, nine converter temperature signals, three phase load current signals, two V_{pcc} signals…… Therefore a very powerful central controller is needed. Only one DSP+ one FPGA can not satisfy what we need. Now we add two more FPGA to the central controller. One FPGA we call it "Modulation FPGA" the other FPGA we name it "Sensor FPGA". Now the central controller has one DSP (6713 DSK) one Xilinx XCV300 FPGA and two Altera Flex 10k30A FPGA.

4.2.1 Modulation FPGA

The modulation FPGA has three functions. The first is to read the three phase switching states from DSP and send them to local controllers. DSP calculates the ABC three phase switching states according to the close loop control strategy. For 7-level CMC, DSP needs to send out 9 converters switching states. The communication protocol between Xilinx FPGA and Modulation FPGA is asynchronous series protocol. The series communication data rate is 250k bit per second. If use one fiber for only one converter's switching state then nine optical fibers are needed. This will increase the cost and cause troubles. If send out all nine switching states in only one optical fiber, the delay issue will be more obvious. For seven level the data rate should be 250k/ (18*9) =1.54 kHz. For higher level the delay is much more serious. We decide to use three fibers for three phase switching states. We modulate all the switching states signals of modular converters in the same phase to one series signal and send them by one fiber. Then we can have enough capacity for future update.

After Modulation FPGA receives the switching states in series format, it needs to decode them and send to all local controllers respectively. For example, for 7-level, there are nine converters. The output signals to all local controllers are 18. Just like Fig. 4.2 shows.

The second function is to send STATCOM feedback to HMI. Two STATCOM status signals are sent out from Xilinx FPGA to Modulation FPGA. Then through the PCI bus the HMI will receive these signals and displays the status of the system in HMI monitor window. The figure below shows the functionality of the modulation FPGA.

The last function of Modulation FPGA is to send the HMI control command to Xilinx FPGA. The control commands are Q-channel reference current and push button commands. These control signals are also modulated to a series communication signal. Fig. 4.3 shows the detail functionality and data path of Modulation FPGA.

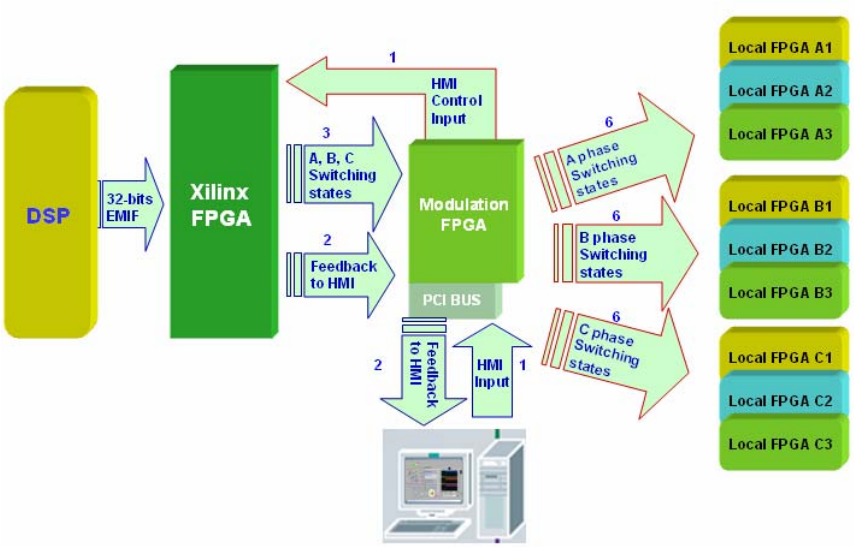


Figure 4.3. Functionality of the Modulation FPGA

The figure 4.4 shows the hardware of modulation FPGA. It's an Altera FLEX 10k30A FPGA. This FPGA has 30,000 logic gates. Compared with Xilinx XCV300 300,000 gates, it's a lot cheaper and easier for implementation. It's a good choice to use this low cost FPGA as the complement for the central controller. The FPGA board provides 40 digital I/Os. All of these I/Os can be programmed as input or output.

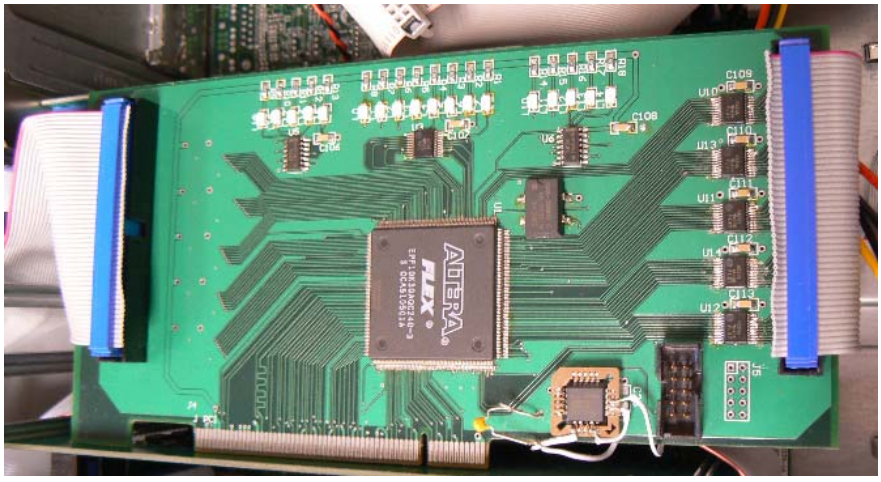


Figure 4.4. Modulation FPGA

4.2.2 Sensor FPGA

The Sensor FPGA hardware is the same as Modulation FPGA, also Altera FLEX 10k30A. But it's reprogrammed for collecting local controller sensor information and

modulates them into one optical signal and sends it to Xilinx FPGA. For 7-level CMC there are nine local controllers.

The sensor signals are sent one by one. For example, for 7-level, in each phase there are three DC voltage signals three current and temperature signals. We need to send them one by one because there is only one optical link for all of sensor signals.

Figure below describes the detail data path from sensor to DSP. Sensor signals go to Local FPGA and are transmitted to Sensor FPGA through series communication. There are 9 channels for a 7-level CMC converter. Then sensor FPGA groups them into one series signals and sends them to Xilinx FPGA. DSP will access these data through EMIF.

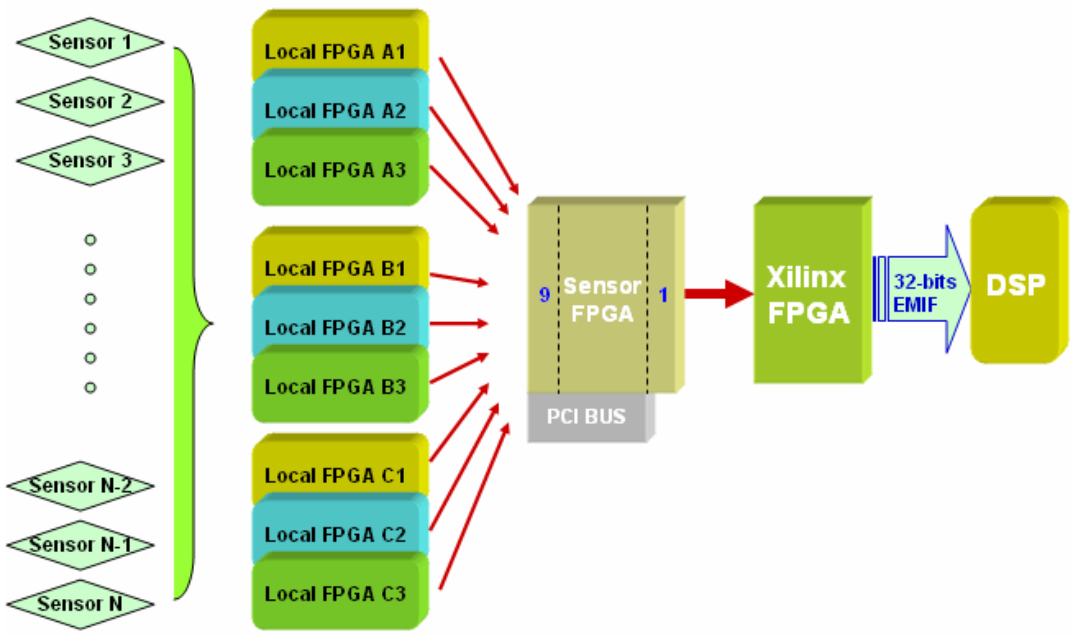


Figure 4.5. Data path from sensor to DSP

4.2.3 Local Controller

Local controller is the “brain” of each module converter. One function of local controller is to collect sensor signals and sends them back to the Sensor FPGA. In the generation four ETO, current sensor, voltage sensor and temperature sensor are integrated. These sensors

send out the sensor signals in digital format. Therefore there is no need of A/D converters in the local controller. The local controller just record this digitalized signals and sent them to sensor FPGA. Finally in DSP, these digital signals will be calibrated to restore their analog signals again.

Another function is to decode the switching state. The switching states for one converter are only two signals. However there are four devices in one module converter. Therefore the Local controller needs to decode them into four gating signals and sends them to the switches on the converter. Figure 4.6 shows the local controller picture. In the board there are 12 ports reserved for the sensor signals in the future. These sensor signals are grouped and modulated into one series signal and sent from the “Communication port to sensor FPGA”. The “Receiving ports from Modulation FPGA” are used to receive switching states signals. Then the switching state signal is decoded into four gating signals and transmitted by the optical transmitter as the picture shows.

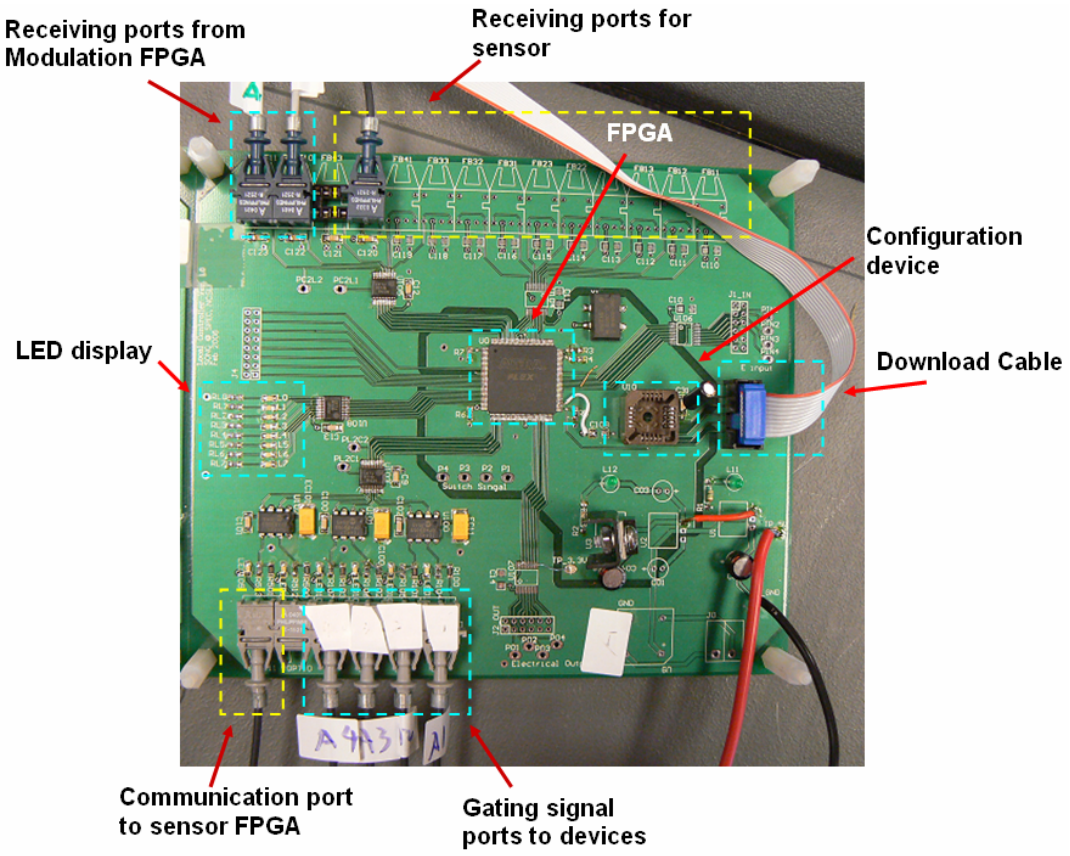


Figure 4.6. Local controller

In the future the local controller needs to realize the fault protection functions, for example over load current protection, over DC voltage protection and over temperature protection. In case of fault, the local controller will quarantine this module converter and send back fault signals to tell DSP what happens.

4.3 Controller Verification Experiment

To verify the controller, the best way is experiment. This section talks about a scale down three-level CMC-based STATCOM system, which uses the digital modular controller proposed in this chapter.

4.3.1 Power Stage

We only need to verify the controller strategy. Therefore a scale down system has been built as the TESTBED. Figure below shows the power supply for this test. It provides three phase 0~120V AC voltages and maximum 5A output current. We will use 50V as our V_{pcc} voltage and the rated current is 4A.

Figure 4.7. AC Power supply for the experiment.

Now we will choose DC bus voltage. Assumed the modulation index at standby mode is 0.7. The line to line voltage is 50V. According to

$$M \cdot V_{dc} = V_{pcc\_phase\_peak} \qquad \text{Equation 4-1}$$

Where M is modulation index, $V_{pcc\_phase\_peak}$ is the phase voltage peak value. Therefore, the DC bus voltage can be calculated. It's 58.3V.

The module converter (H-bridge) is constructed with an Intelligent Power Module (IPM) PM50RSA120. The IPM is an isolated base module designed for power switching applications operating at frequencies to 20 kHz. Built-in control circuits provide optimum gate drive and protection for the IGBT and free-wheel diode power devices. The protection includes: Short Circuit, Over Current, Over Temperature and Under Voltage. The IPM maximum block voltage is 1200V, and the maximum current rating is 50A. The Fig. 4.8

shows the picture of a module converter for this experiment. There is a driver board to transfer optical gating signals to electrical signals.

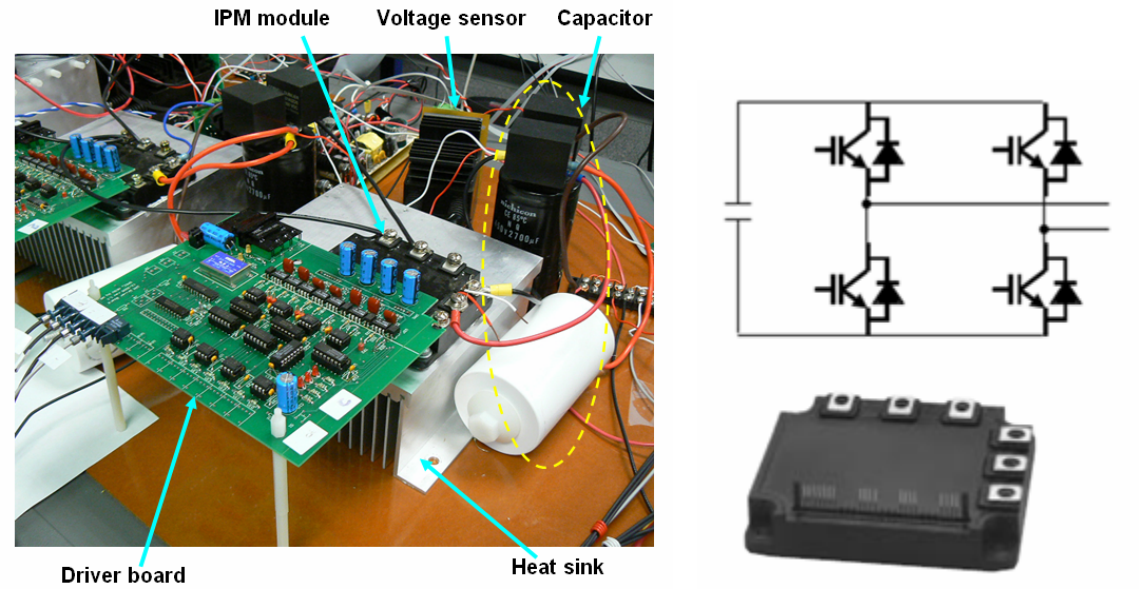


Figure 4.8. Module Converter H-bridge and IPM

For this small scale test a 2.5 mH inductor is large enough as the coupling reactance. We use 5.4mF capacitor for the DC bus. Finally all the parameters for the scale down test are shown in the table below.

Table 4-1. Test parameters

Three-level CMC-based STATCOM	3-phase 3-wire Wye-connection
Individual DC bus voltage	58.3V
Rated RMS reactive current	5A
Capacitor Impedance	5.4mF /Phase
Coupling Reactor impedance	2.5mH /Phase
PCC Line Voltage	50V

4.3.2 Optimal Combination Modulation Strategy

As described in [23] [24], the optimal combination modulation strategy works with low switching frequency and has low current harmonic distortions. Fig. 4.9 shows the optimal combination modulation strategy for the seven-level cascade multilevel inverter. V_{H1}, V_{H2} and V_{H3} are output voltages of three H-bridges each phase. As shown in Fig. 4.9, the output voltage of an H-bridge has five switching angles in the period from 0 to π/2. So the phase voltage of the inverter can achieve fifteen switching angles in the period from 0 to π/2. Based on Fourier series transformation, the harmonics of V_{H1} can be expressed as:

$$V_{H1(n)} = \frac{4U_D}{n\pi}[\cos(n\theta_1) - \cos(n\theta_2) + \cos(n\theta_3) - \cos(n\theta_4) + \cos(n\theta_5)] \qquad \text{Equation 4-2}$$

where, U_D is the voltage of dc capacitor of an H-bridge. For a three-phase application, the triple-order harmonics in each phase need not be canceled as they automatically cancel in the line-to-line voltages. Here, the 5th, 7th, 11th and 13th order harmonics are chosen to be removed. The switching angles of the first H-bridge must satisfy the following equations:

$$\begin{aligned}
V_{H1(1)} &= \frac{4U_D}{\pi}[\cos(\theta_{11}) - \cos(\theta_{12}) + \cos(\theta_{13}) - \cos(\theta_{14}) + \cos(\theta_{15})] \\
0 &= \cos(5\theta_{11}) - \cos(5\theta_{12}) + \cos(5\theta_{13}) - \cos(5\theta_{14}) + \cos(5\theta_{15}) \\
0 &= \cos(7\theta_{11}) - \cos(7\theta_{12}) + \cos(7\theta_{13}) - \cos(7\theta_{14}) + \cos(7\theta_{15}) \\
0 &= \cos(11\theta_{11}) - \cos(11\theta_{12}) + \cos(11\theta_{13}) - \cos(11\theta_{14}) + \cos(11\theta_{15}) \\
0 &= \cos(13\theta_{11}) - \cos(13\theta_{12}) + \cos(13\theta_{13}) - \cos(13\theta_{14}) + \cos(13\theta_{15})
\end{aligned} \qquad \text{Equation 4-3}$$

After solving these equations and applying some conditions [23] [24], a set of angles respect to different modulation index can be calculated. These data are verified in the simulation of 10 MVAr STATCOM system [24].

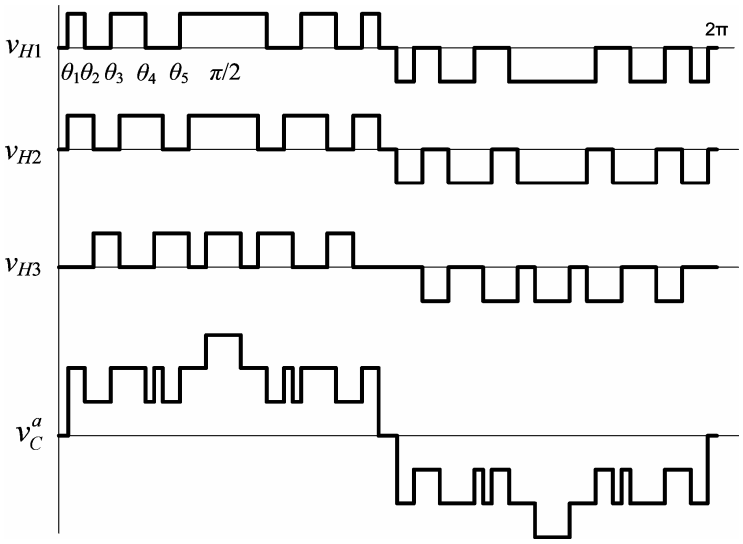


Figure 4.9. Optimal PWM modulation strategy.

To implement this method in DSP, a function named Optimal_PWM () is created in file AED_106_TVA.c. Please refer to the appendix for more details.

4.3.3 Compensator Design

This is a three-level CMC. According to the system parameters and the small signal transfer function discussed in chapter 2, we can draw the bode plot for the transfer functions and design the compensators.

I. ***Design of Current Loop Compensator, H_{id}.***

According to the transfer function of G_{idd} in equation 2-10,

$$G_{idd} = \frac{\tilde{i}_d}{\tilde{d}_d} = \frac{NE(R_s + L_s S)}{L_s{}^2 S^2 + 2L_s R_s S + R_s^2 + \omega^2 L_s^2}$$

Where N =1, R_s = 0.15Ω, L_s = 2.5mH, E = 58.3 and ω= 377. Then the bode plot of the open loop control-to-output-current transfer function G_{idd} is shown below:

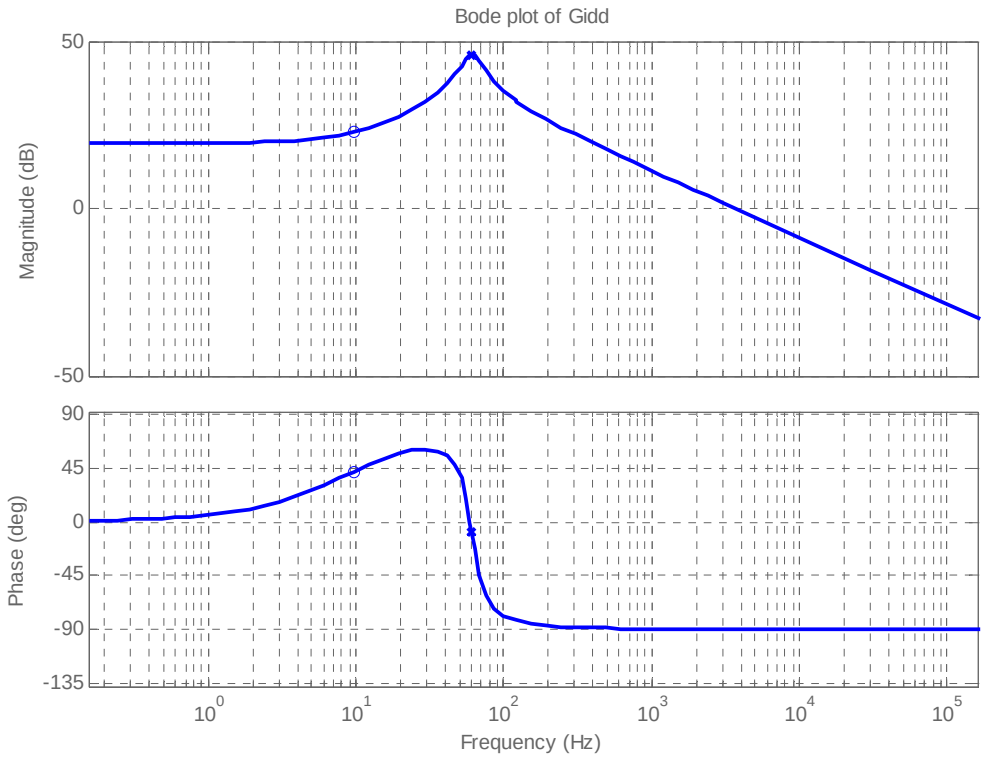


Figure 4.10. Bode plot of open-loop control-to-output-current transfer function $G_{idd}(s)$

Among well-known compensators, the lag compensator or the proportional-plus-integral (PI) is the best candidate based on its simplicity and reliability. The D-channel current loop compensator $H_{id}(s)$ is a PI controller. With the compensator, the closed- loop transfer function of the D-channel current can be expressed as follows:

$$T_{id}(s) = H_{id}(s) \cdot G_{idd}(s) \qquad \text{Equation 4-4}$$

where $H_{id}(s) = K_P + \frac{K_i}{s}$. After the optimization process, the best-designed parameters for the PI compensator are given in Table 4-3. The current loop gain is plotted, as shown in Figure 4.11, to verify its stability. The desired crossover frequency is selected to be as high as possible, but must be less than 600 Hz, which is the effective switching frequency of OPWM. Because there are only ten pulses in a fundamental period, the designed crossover frequency is 87.7 Hz, with reasonably large phase margin of 96.2 degrees. The characteristics

of current loop gain $T_{id}(S)$ are listed in Table 4-3. Due to identical transfer function, the Q-channel-current loop compensator $H_{iq}(S)$ is designed as $H_{id}(s)$.

Table 4-2. Designed PI compensator parameters and current loop gain characteristics.

Parameters	Values
PI Compensator Hid(s)	
K_P	0.013
K_i	1.3
Loop Gain $T_{id}(s)$ Characteristics	
Crossover Frequency (Hz)	87.7
Phase Margin (degrees)	96.2

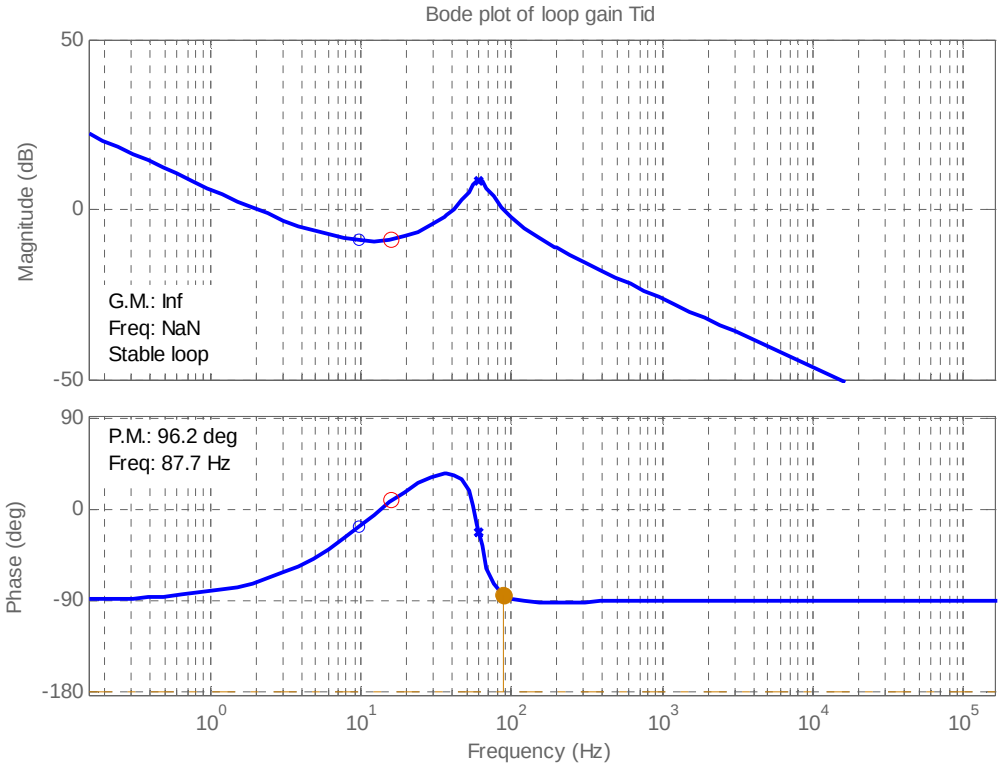


Figure 4.11. Bode plot of the loop gain $T_{id}(s)$

II. *Design of DC Capacitor Voltage Compensator, H_{Ed},*

With the D-channel current loop closed, the new transfer function, i.e., the reference current-to-DC-voltage transfer function, $T_{Eid}(S)$, is as follows:

$$T_{Eid}(s) = \frac{E}{i_d^*} = G_{Eid}(s) \cdot \frac{T_{id}(s)}{1 + T_{id}(s)} \qquad \text{Equation 4-5}$$

where $G_{Eid}(S)$ is open loop output-current-to-DC-bus-voltage transfer function as shown in equation 2-14. Then we can draw the block diagram of the DC voltage loop.

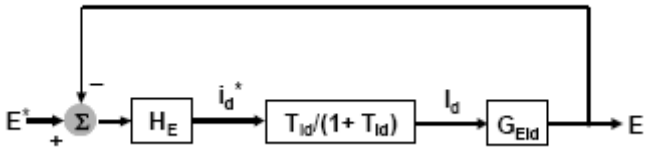


Figure 4.12. DC voltage loop block diagram [9]

The bode plot of the transfer function $T_{Eid}(s)$ is shown in Figure 4.13.

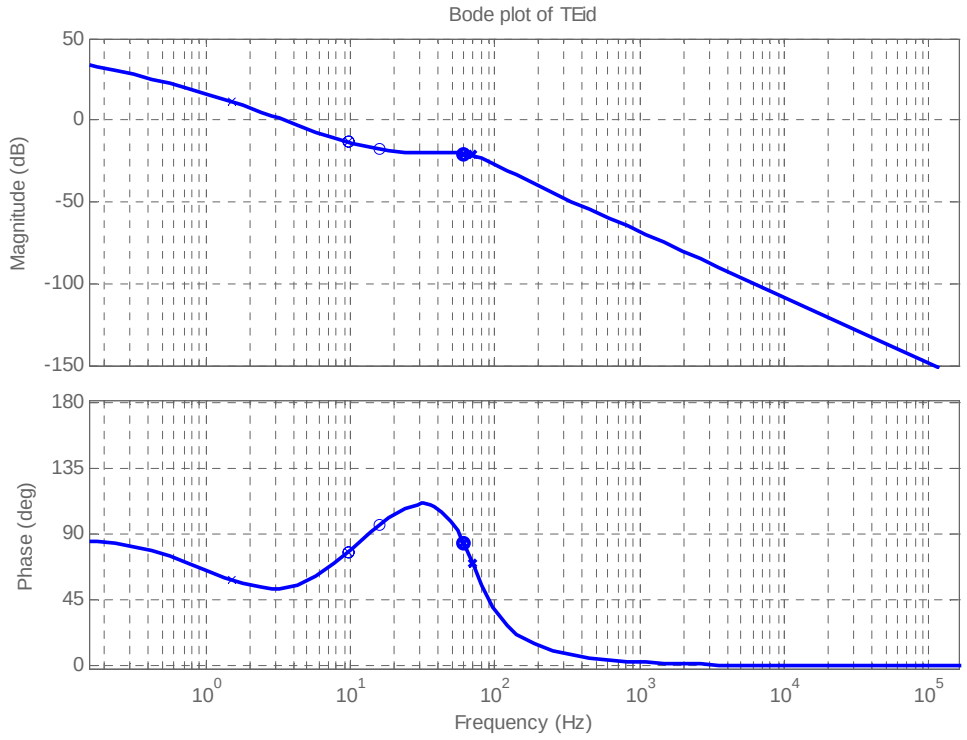


Figure 4.13. Bode plot of current-to-DC voltage transfer function $T_{Eid}(s)$

The crossover frequency should be as high as possible; however, it needs to be limited at a frequency that is reasonably lower than the crossover frequency of the current-loop gain in order to avoid interferences between these two control loops. In this design, the crossover frequency of the voltage-loop gain is placed at 4.69 Hz, which is almost twenty times lower than that of the current-loop gain. The corresponding phase margin is 47 degrees. The parameters of compensator $H_{Ed}(s)$ and the characteristics of the close-loop D-channel *loop* gain $T_{Ed}(s)$ are shown in the table 4-4. The bode plot of the close-loop voltage loop gain is shown in Figure 4.14.

Table 4-3. Voltage loop compensator parameters and the characteristics of the voltage loop

Parameters	Values
PI Compensator Hid(s)	
K_P	1.656
K_i	8.28
Loop Gain $T_{Ed}(s)$ Characteristics	
Crossover Frequency (Hz)	4.69
Phase Margin (degrees)	47

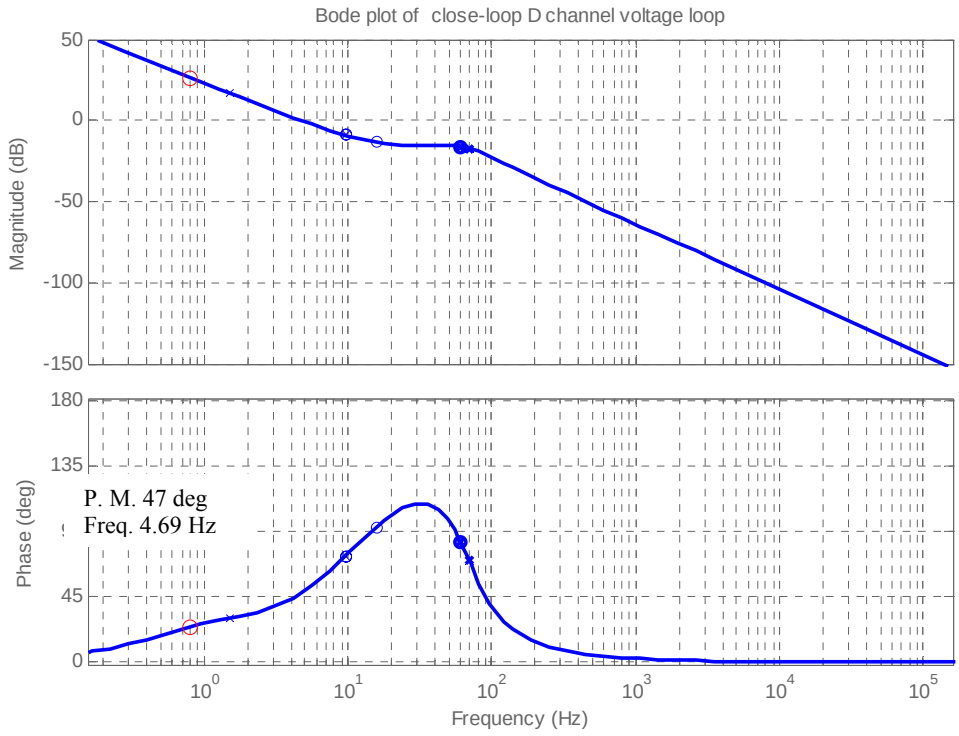


Figure 4.14. Bode plot of close-loop D channel voltage loop

4.3.4 Individual DC Bus Voltage Balance Loop Design

From the control block diagram of the CMC-based STATCOM in Fig. 3.5, we can see that only the average DC voltage is being controlled. This control scheme assumes that all the module converters have same loss. Then the individual DC bus voltages should be identical. However, as we know, there are no total identical things in this world. Therefore, if only average DC voltage is controlled, the individual DC voltages maybe unbalance.

In [25] Dr. Fangzheng Peng proposed a solution. An individual DC voltage loop is created for each module converter and generates a small phase shift angle $\Delta\alpha_{ci}$ to compensate the power loss of the module converter. Figure 4.17 (a) shows the control scheme of a multilevel cascaded inverter, which has individual DC voltage loop. Where the current direction is inversed compared with the STATCOM system we discuss here. In [25], fundamental switching modulation strategy is used. But we can extend this individual DC

loop idea to other modulation method for example SPWM or OPWM. In Fig. 4.17 (b), the principle of the individual DC voltage loop is described. The switching angles, θ_i [i = 1, 2. . . (M-1)/2], are calculated off-line to minimize harmonics for each modulation index, just like OPWM we discuss above. Since the phase current i_{Ca} is always leading (capacitive mode) or lagging (inductive mode) the phase voltage V_{Ca}, by 90°, the average charge to each dc capacitor is equal to zero over every half line cycle. The average charge to each dc capacitor over half cycle [0, π], can be expressed as

$$Q_i = \int_{\theta_i}^{\pi-\theta_i} \sqrt{2} I \cos\theta d\theta = 0 \qquad \text{Equation 4-6}$$

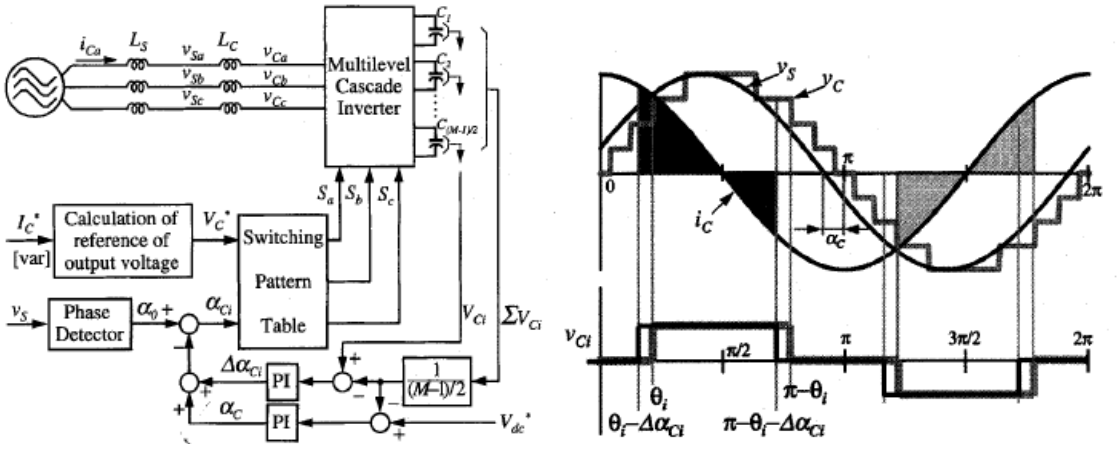
From Fig. 4.17 (b) we can see that if create a small phase shift $\Delta\alpha_{ci}$, the charge balance of the capacitor will be destroyed. The average charge to each dc capacitor over half cycle can be expressed as

$$Q_i = \int_{\theta_i-\Delta\alpha_{ci}}^{\pi-\theta-\Delta\alpha_{ci}} \sqrt{2} I \cos\theta d\theta = 2\sqrt{2} I \cos\theta_i \sin\Delta\alpha_{ci} \qquad \text{Equation 4-7}$$

The average charge is proportional to $\Delta\alpha_{ci}$ when $\Delta\alpha_{ci}$ is small. Therefore each module converter's DC voltage can be controlled by slightly shifting the switching pattern. For high-voltage high-power applications, the total power loss of the inverter is typically less than 1%, thus $\Delta\alpha_{ci}$ << 0.01 rad. From equation 4-7 we can also see that if the current is very small, for example in stand-by mode, the Q is also very small. Then it's very hard to balance the DC voltage even though we have small phase shift angle.

In Fig. 4.15 (b), the inverter is working at capacitive mode. The inverter is outputting reactive power to the grid. In this figure if the $\Delta\alpha_{ci} > 0$, the capacitor will be charged, if $\Delta\alpha_{ci} < 0$, the capacitor will be discharged. When the inverter is working at inductive mode,

the current lags voltage 90 degree. When $\Delta\alpha_{ci} > 0$, the capacitor is discharged. When $\Delta\alpha_{ci} < 0$, the capacitor is charged. This means during different working mode the phase shit need to be inversed to charge or discharge capacitor.



(a) (b)

Figure 4.15. (a) Control block diagram with individual DC loop. (b) Control principle for each DC capacitor voltage. [25]

4.3.5 Updated Control Scheme and Off-line Simulation

To add the individual DC voltage loops, new control blocks are added to the original control block diagram shown in Fig. 3.5. The updated control block diagram is shown in the Figure 4.16 below. Now we assume the current direction is from the STATCOM to the Grid. Therefore when I_{qref} <0, STATCOM is in capacitive mode.

The input of Individual DC loop compensator H_{DCi} are errors of each module converter's DC voltages. The outputs of the H_{DCi} are the small phase shift angles. The polarity of the phase shift angle determines to charge or discharge the capacitor. In inductive mode and capacitive mode, the angle polarity is inversed. Therefore, there is a MUX behind the compensator H_{DCi} to determine the polarity of the phase shift. In capacitive mode, I_{qref} <0, the output of the MUX is 1; in inductive mode, I_{qref} >0, the MUX output is -1.

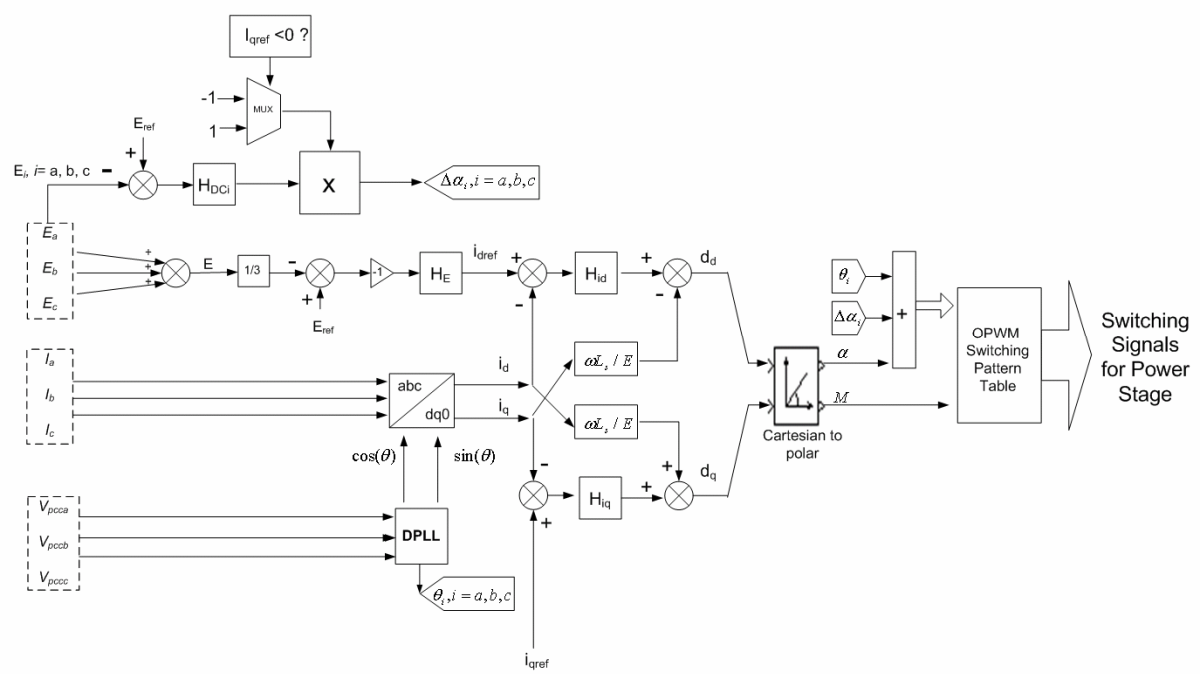


Figure 4.16. Close-loop control block diagram of a 3-level CMC based STATCOM

To verify the individual DC loop and the designed compensators' parameters, off-line simulation using Matlab SIMULINK has been done. Figure 4.17 shows the power stage of a three-level CMC STATCOM IN SIMULINK. All the simulation parameters are the same as table 4-2, 4-3 and 4-4.

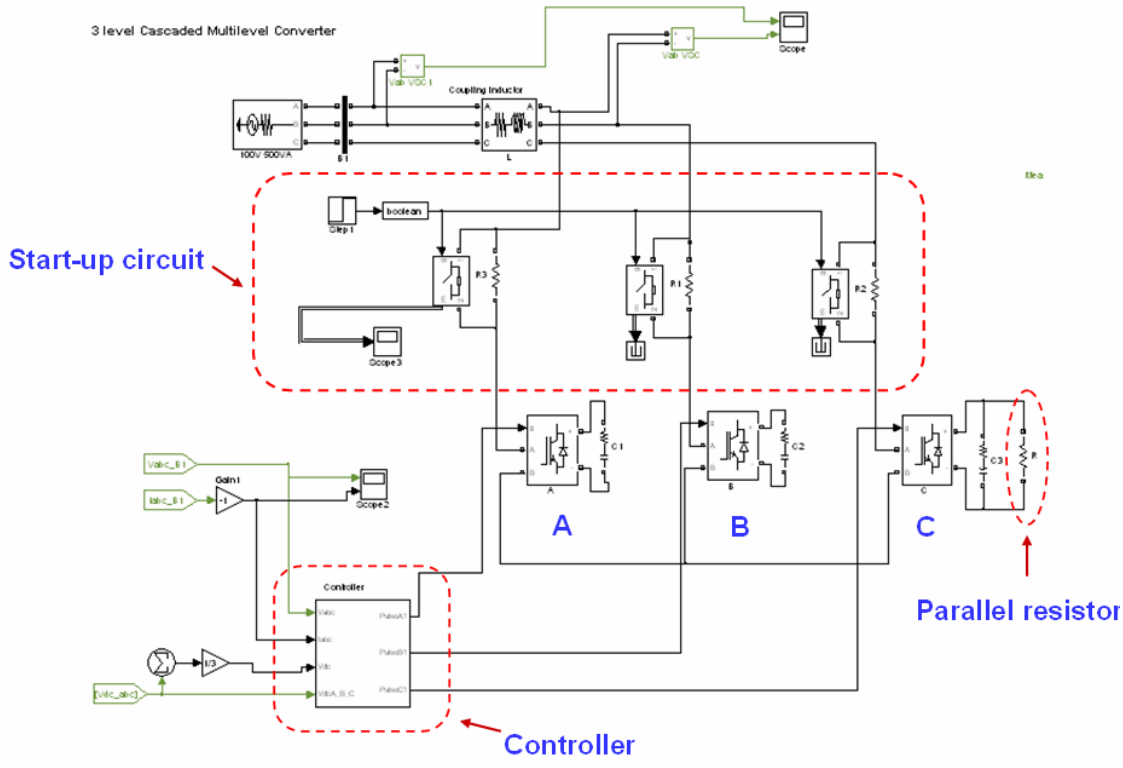


Figure 4.17. Three-level CMC STATCOM simulation block diagram

To create a DC voltages unbalance situation, a 500 ohm resistor is paralleled with C phase capacitor, which will cause additional power loss of C phase H-bridge. The simulation results are shown below. The range of the small phase shift angle is within [-0.01 rad, 0.01 rad].

When there is no individual DC balance loops in the simulation, Fig. 4.18 shows the Q-channel current and Fig. 4.19 shows the three phase DC voltages. Before 1.7 seconds, the Q-channel current reference is 0A. The STATCOM works at stand-by mode. At this time we can see that V_{DCa} is equal to 59V; V_{DCb} is euqual to 58.3V; V_{DCc} is equal to 57.5V.and after 1.7 seconds I_{qref} jumps to 7A. STATCOM works at inductive mode. After 3 seconds, I_{qref} steps down to -7A. STATCOM works at capacitive mode. From Fig. 4.23 we can see that, because the C phase power loss is always bigger than the other phase, C phase DC voltage is

always the lowest one. No matter what mode the STATCOM is working at, the three DC voltages are always unbalance.

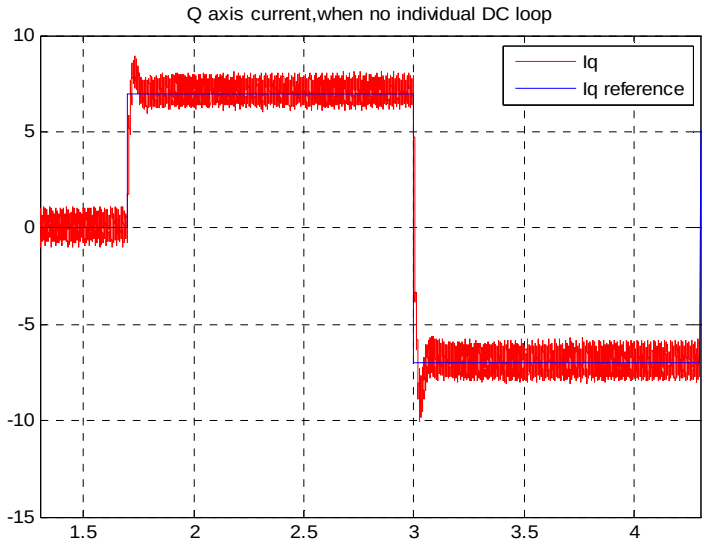


Figure 4.18 Q-channel current, when there are no individual DC loops.

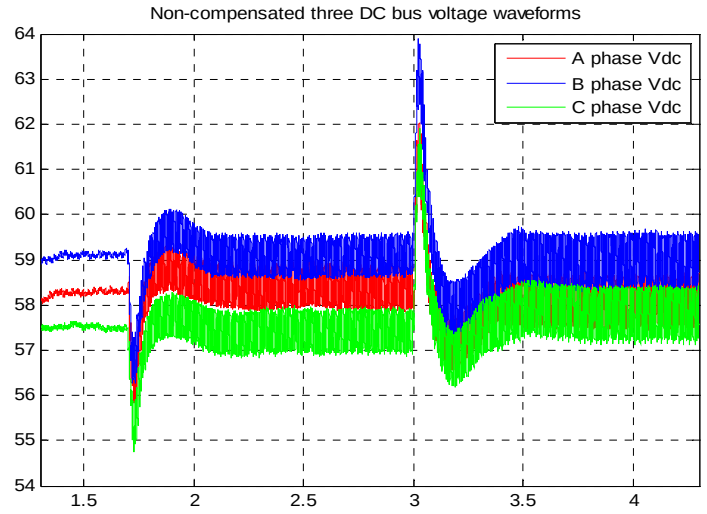


Figure 4.19. Converter DC voltages when there are no individual DC loops.

The Fig. 4.20 and Fig. 4.21 shows the simulation results when the DC voltages are compensated by individual DC voltage loops. Compare with the simulation results above, we can see that there is no big difference in the Q-channel current loops. Because the small phase shift angle only affect DC voltages and have little effect in the I_q. Also, in stand-by

mode, because the current is very small, almost zero, according to equation 4-7, the average charge Q is almost zero too. Therefore, in stand-by mode the DC voltages almost can't be compensated. But during 7A capacitive mode and 7A inductive mode, the three DC voltages are compensated perfectly. The simulation results verify the new control scheme and the compensator parameters.

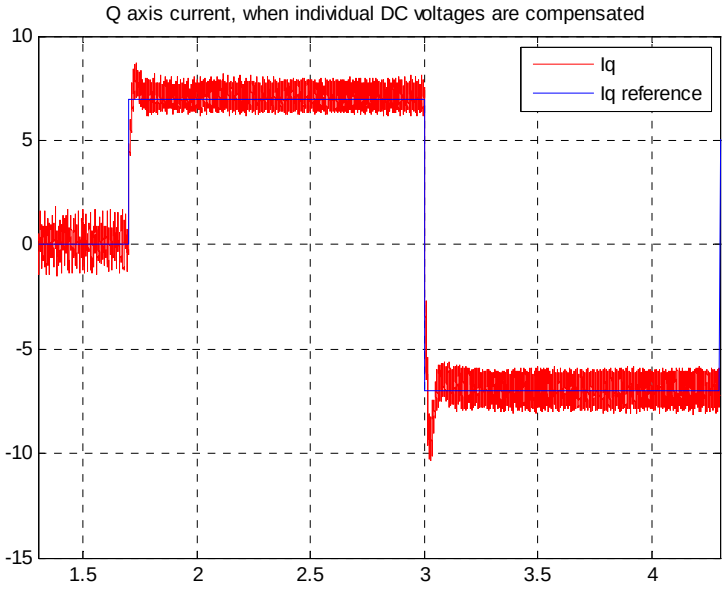


Figure 4.20. Q-channel current with individual DC loops

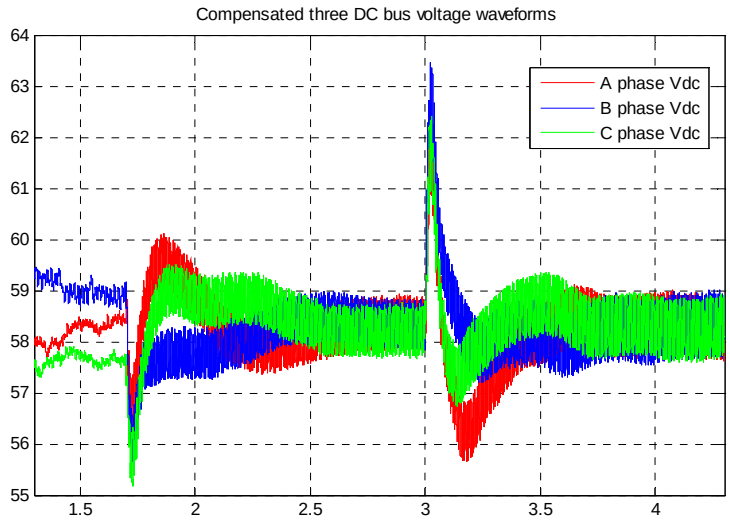


Figure 4.21. Converters' DC voltages with individual DC loops

4.3.6 50V 5A Experiment

I. ***System setup***

The experiment setup is shown in figure 4.22 and 4.23. The system includes a 50V 5A AC power supply, three module converters H-A, H-B, H-C, interface boards, central controller and three local controllers. The central controller includes a Ti's TMS320c6713 DSK board, a Xilinx XCV300 FPGA, Modulation FPGA and Sensor FPGA. Because the Modulation FPGA and Sensor FPGA communicate with PC through PCI bus, they are plugged in PC's PCI slots.

There are also five voltage sensors. Two of them measure line-line voltages V_{ab} and V_{bc}, the others measure the three DC voltages of the module converters. There are also three current sensors to detect load currents.

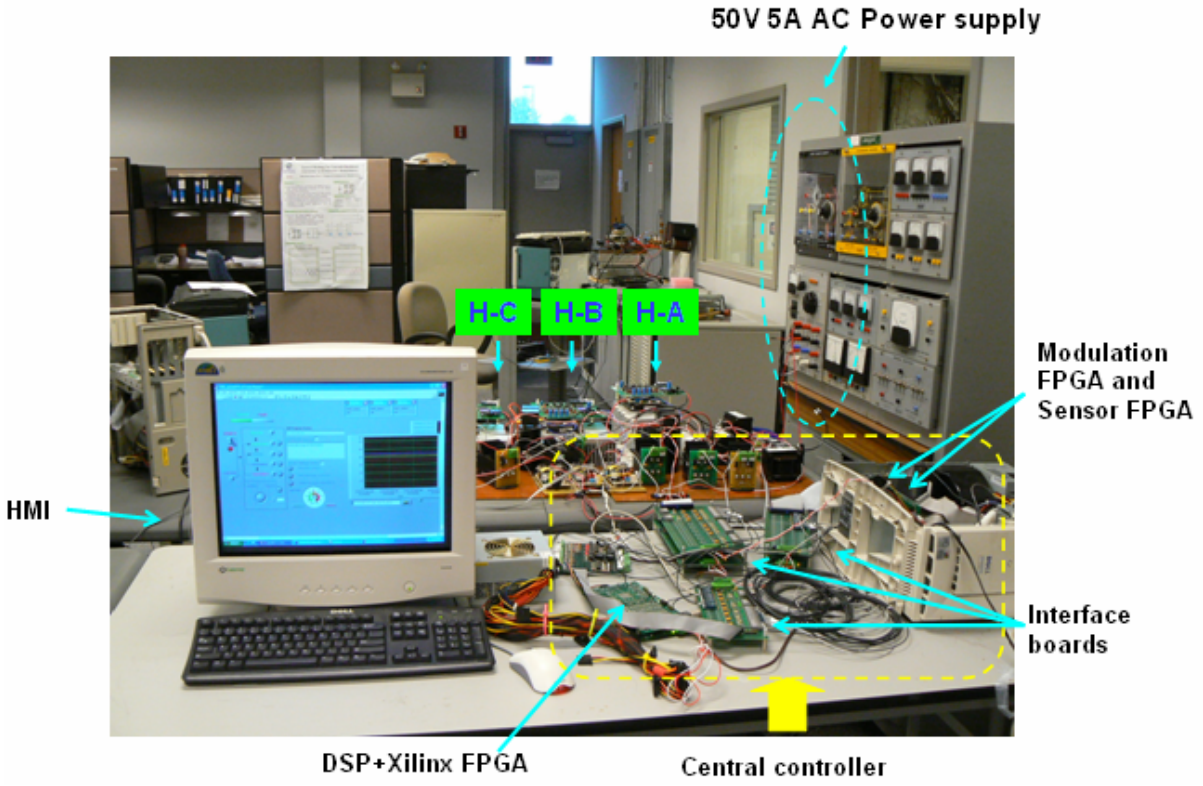


Figure 4.22. Experiment system front view

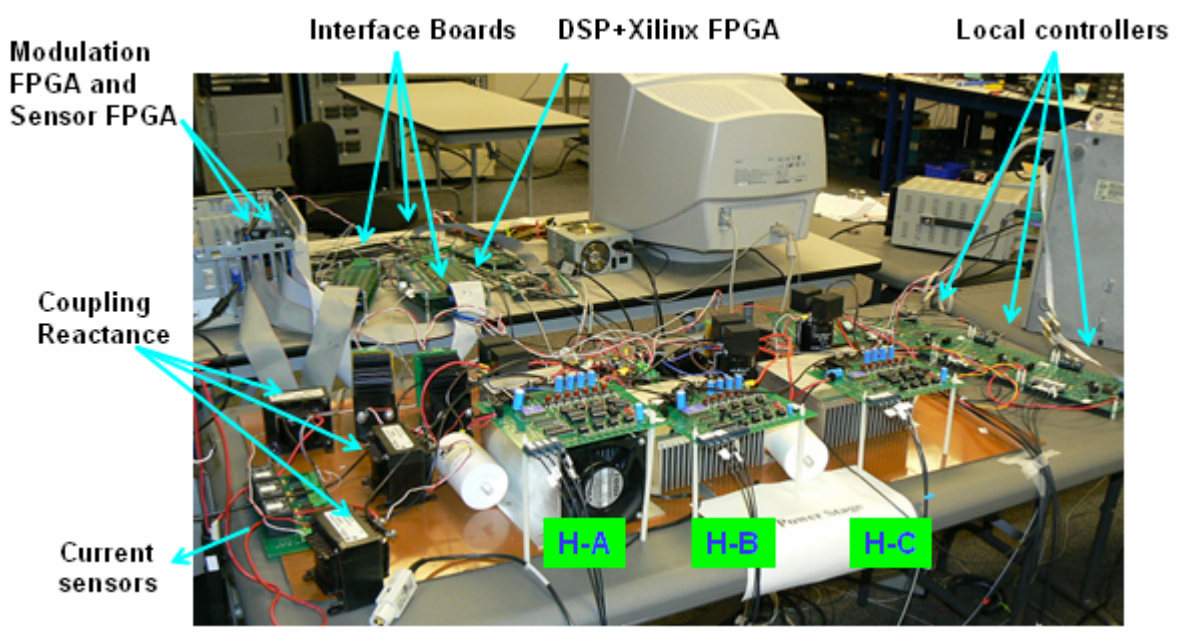
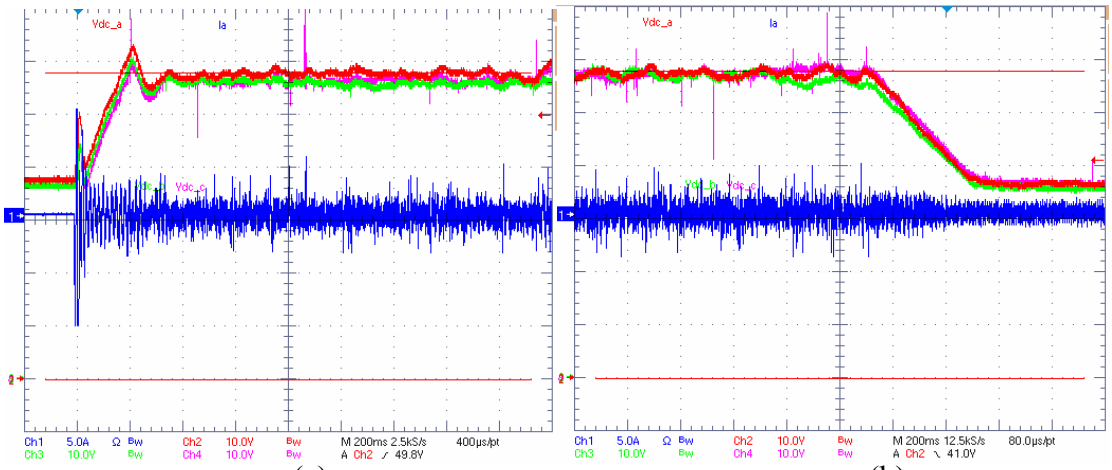


Figure 4.23. Experiment system back view

II. ***Charge and Discharge Test results***

Because the line-line voltage is 50V, therefore after diode mode charge, the DC voltages are stable around 35V. After push the "Charge" button in the HMI, the DC voltages are boosted to the reference point 58.3V. During the boost charge process, the I_{dref} is equal to -3A and I_{qref} is 0A. The charge waveforms are shown in Fig. 4.24 (a). The blue line is load current. The green, red and pink lines are three DC voltages. Fig. 4.24 (b) shows the discharge process. Different with charge process, during discharge mode, I_{dref} is equal to 3A and I_{qref} is 0A.

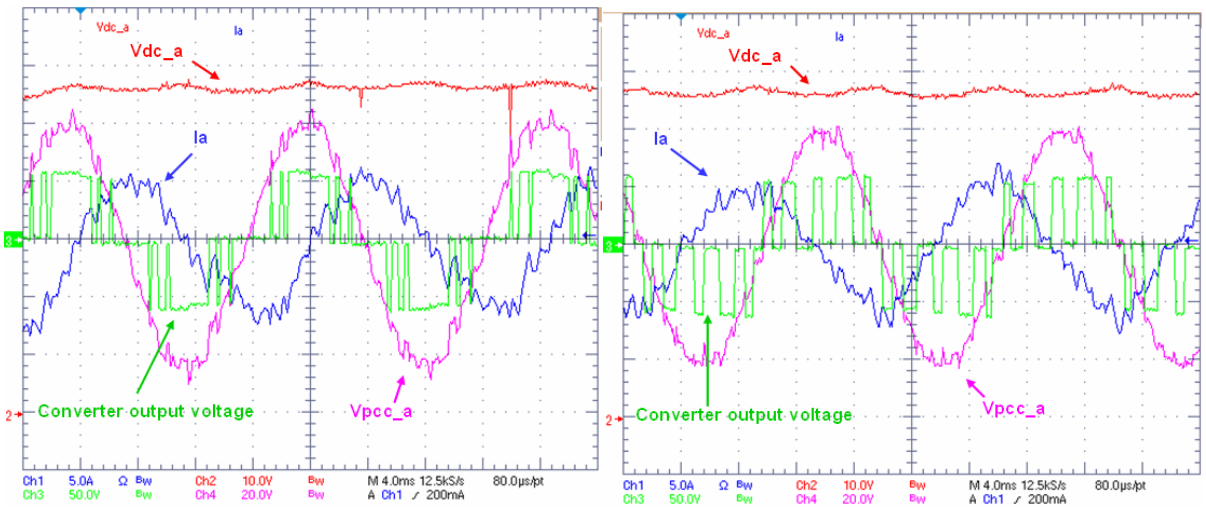


(a) (b)

Figure 4.24. (a) Boost charge; (b) Discharge.

III. ***Steady State Capacitive Mode and Inductive Mode Test Results.***

Fig. 4.25 (a) shows the steady state capacitive mode when I_{qref} is equal to -5A. The pink line is the A phase Vpcc voltage. The red line is A phase DC voltage. The blue line is the A phase load current. And the green line is the converter output voltage. Because the power supply's power rating is very small, it's a weak system. The V_{pcc} voltages have some harmonics. However the system is still running smoothly. This means the DPLL block can sustain such a noisy condition. Also during capacitive mode the current is lagging phase voltage 90 degree and during inductive mode the current is leading 90 degree. From the converter voltage output waveform, we can see that there are always five square pulses during half cycle. They match the OPWM waveform which has been discussed in Fig. 4.9.

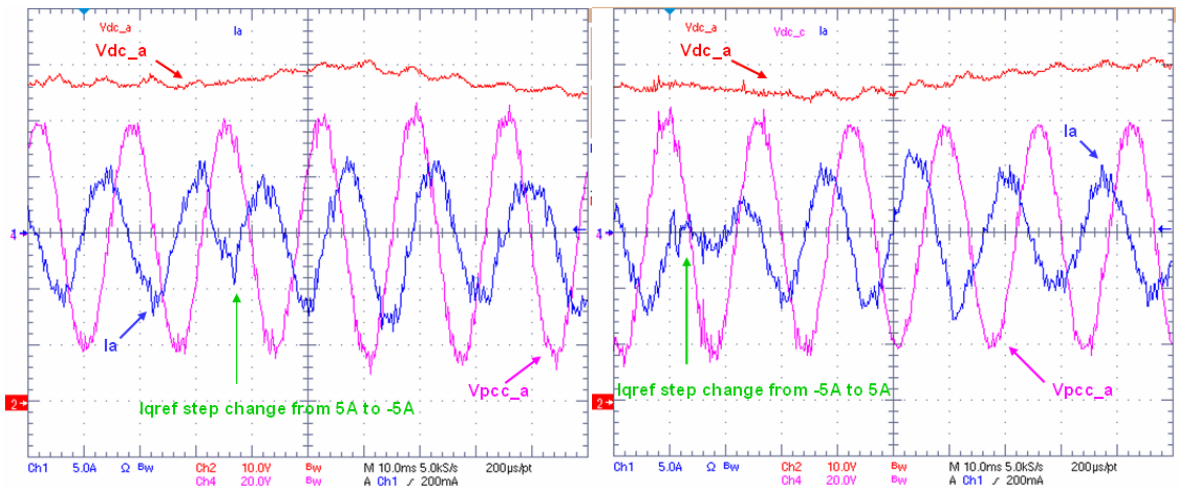


(a) (b)

Figure 4.25. (a) 5A capacitive mode; (b) 5A inductive mode.

IV. ***Dynamic Response***

Here we will step change the I_{qref} and see the transient response of the system. Fig. 4.26(a) shows the step change from inductive mode to capacitive mode. The I_{qref} step changes from 5A to -5A. The green line point out the step change time. We can see that after three cycles the current reach stable state again. Fig. 4.26 (b) shows the transient from capacitive mode to inductive mode. The I_{qref} step changes from -5A to 5A. The current becomes stable after three cycles. These transient results exactly match the current loop design. The current loop bandwidth is designed at 87.7 Hz. Normally, the transient response time is 4~5 times longer than 1/bandwidth. That's 5/87.7 = 57 ms. 57 ms is almost 3 cycles. Therefore the experiment results match the design perfectly.

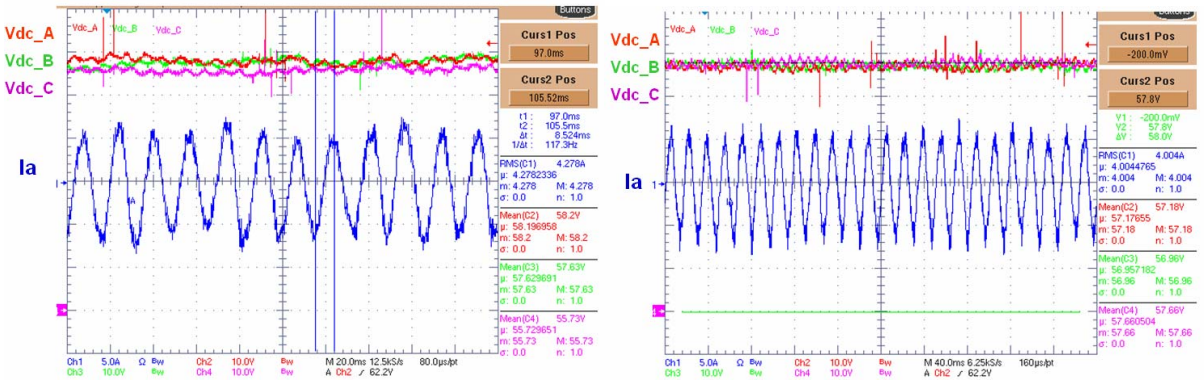


(a) (b)

Figure 4.26. (a) Inductive mode to capacitive mode; (b) Capacitive mode to inductive mode.

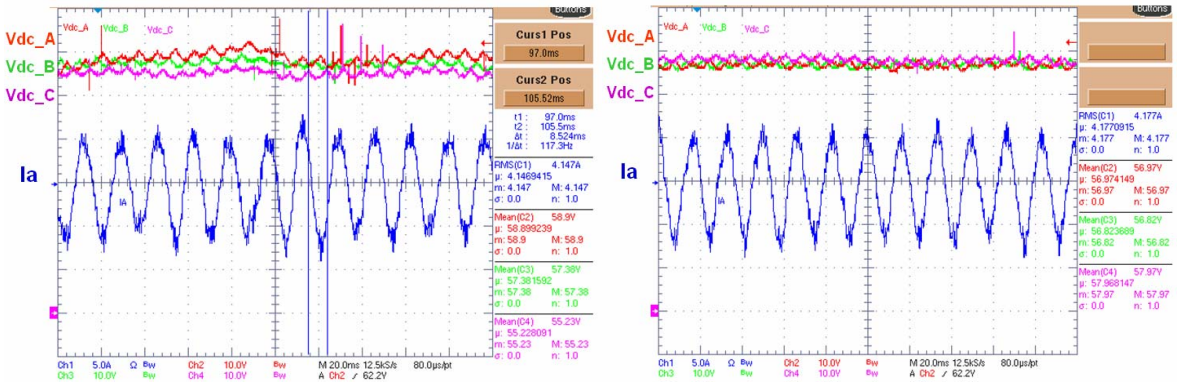
V. ***Individual DC loops test***

To create the DC voltage unbalance, a 6.8 k ohm resistor is paralleled with the C phase capacitor, which creates additional loss (around 0.5 w) and makes DC voltage in the C phase converter lower than that of A and B phase converters. Fig. 4.27 (a) and 4.28 (a) are the experiment results when there are no individual DC loops (IDVL). Fig. 4.27 (b) and 4.28 (b) are the results that the DC voltages are compensated by the individual DC loops. The results show that without IDVL the C phase capacitor voltage is always the lowest. The voltage difference is beyond 2V. With IDVL, the three capacitor voltages are very close. Phase C voltage is not always the lowest. Three DC voltages vibrate around the reference DC voltage. The voltage differences are smaller than 1V. This experiment results truly proved effectiveness of the IDVL during capacitive mode and inductive mode.



(a) (b)

Figure 4.27. (a) 5A capacitive mode without IDVL; (b) 5A capacitive mode with IDVL



(a) (b)

Figure 4.28. (a) 5A inductive mode without IDVL; (b) 5A inductive mode with IDVL

4.4 Conclusion

In this chapter, a modular controller topology for CMC-based STATCOM is proposed. Also an individual DC voltage loop strategy is proposed to balance each module converter's DC voltage. A new modulation strategy called OPWM has been reviewed and implement in the new controller software. This chapter also explains the compensators design for the current loop and voltage loops in details. The simulation results verify the design and 50V 5A experiment has been conducted and discussed. Finally the proposed digital modular controller is verified both in software and hardware.

Chapter 5 Summery and Future research

5.1 Conclusion

The main contribution of this thesis is the development of digital controllers for CMC-based STATCOM system. These designs not only include hardware design but also software design as well as modeling and control theory.

Two digital controllers based on two different architectures have been designed. The first one is the conventional centralized controller topology, which has been shown in chapter three. This controller structure is simple and straight forward. The TI's DSP + Xilinx FPGA architecture provides enough calculation capacity and enough digital I/Os to control the whole system. This controller design provides me the opportunity to understand the insight of the embedded systems design. The software design is the key to implement the control strategy. In chapter three, detail software design procedures are discussed, including traditional PI controller, low pass filter and Digital Phase Lock Loop design. The Real Time Data Exchange technology provides us an easy way to build a Human Machine Interface, which is more cost effectively compared with conventional HMI design. The 480V 100A experiment proves that the controller design is a success.

However, the conventional centralized controller architecture has some disadvantages. It's not easy to expand. If you want to increase the power rating of CMC-STATCOM, a lot of hardware needs to be modified. And only one DSP plus one FPGA may not be powerful enough for controlling such a complex system. Therefore a new topology of controller is proposed in chapter four. The proposed modular digital controller has one central controller and multiple local controllers. Compared with the conventional one, the new architecture has a lot of advantages. Firstly, it has more powerful calculation capacity and more digital I/O.

The central controller include a TI's TMS320c6713 DSK board, a Xilinx 300,000 gates FPGA, two Altera 30,000 gates FPGA. Compared with the conventional one, the FPGA share a lot of works from DSP and provide almost five times more digital I/O. Secondly, the local controller provides a "brain" for the H-bridge. The CMC-based STATCOM with a modular digital controller becomes truly modular because of the local controller. The central controller only needs to provide simple switching states. The local controller will take care of the rest of the work, including dead time generation, over voltage, over-current and over-temperature protections as well as sensor signals collection. Also with the help of series communication technology, the number of wires is greatly reduced. The optical links in this design provide better noise susceptibility. As a summary, this control structure increases the modularity, system reliability and reduces noise interference to the system. To balance each DC voltages, the individual DC voltage balance loops are successfully implemented in the DSP software. The simulation and experiment results prove it's effectiveness in capacitive mode and inductive mode.

5.2 Limitations

In the developed Human Machine Interface for the 4.5 MVA STATCOM system, we used the RTDX technology and simplified the design. There is no need for any additional hardware. However, it also has some disadvantages. Because the RTDX needs to call the Code Composer Studio in the PC to communicate with the DSP, the reliability depends on the Code Composer Studio. This means if the Code Composer Studio is interfered, we will lose control of the DSP. Unfortunately, the Code Composer Studio is not very stable and it sometimes terminates abnormally. Finding a solution to this problem is needed in the future if RTDX needs to be used for practical applications.

Another limitation is the noise interference issue. The 6713DSK board is a powerful DSP board, however it was designed for digital processing, e.g. video signal audio signal processing. Therefore there is no shielding protection on the board. The STATCOM is a high power application. During the operation, the device switching and the AC current through the coupling inductor generate a lot of Electromagnetic interference (EMI). The high EMI interferes the emulator on the 6713DSK board. This emulator is used for the communication between the DSP and PC.

5.3 Future Work

Digital modular controller is a very attracting research topic. There are still a lot of things to investigate in this area.

Firstly, more functions can be added to the local controller. For example more protection functions, converter diagnostic functions and one can even use local controller to monitor the power supply and sensor working status.

Secondly, a new HMI which is independent of CCS is needed to improve the stability issues. The Altera FPGAs in the central controller are using PCI bus, which are good enough to communicate with the PC directly. With the help of the PCI bus, we don't need RTDX and CCS anymore. This will greatly improve the system reliability.

What is more, we only test the system in a three-level small scale system. In the future we should add more module converters to test the proposed architecture and communication protocol.

Finally, there are still some practical issues when the modular controller is actually mounted on a real STATCOM. Power supply of the local controller should be considered. Also the switching noise interference to the local controller should be avoided.

REFERENCES

1. Understanding FACTS Concept and Technology of Flexible AC Transmission Systems, NARAIN G. HINGORANI, LASZLO GYUGYI.
2. Sumi Yoshihiko, Harumoto Yoshinobu, Hasegawa T, et al. "New Static VAR Control Using Force-Commutated Inverters", IEEE Trans. On Power Apparatus and Systems. 1981, PAS-100 (9): 4216-4224.
3. Arsoy, A., Yilu Liu, Shen Chen, et al, "Dynamic performance of a static synchronous compensator with energy storage", Power Engineering Society Winter Meeting, 2001, IEEE, Volume: 2, 28 Jan.-1 Feb. 2001: 605-610 vol.2.
4. Jih-Sheng Lai and Fang Zheng Peng, "Multilevel Converters-A New Breed of Power Converters", Industry Applications, IEEE Transactions on Volume 32, Issue 3, May-June 1996 Page(s):509 - 517 Digital Object Identifier 10.1109/28.502161.
5. Qingguang Yu, Pei Li, Wenhua Liu, Xiaorong Xie, "Overview of STATCOM technologies", Electric Utility Deregulation, Restructuring and Power Technologies, 2004. (DRPT 2004). Proceedings of the 2004 IEEE International Conference on Volume 2, 5-8 April 2004 Page(s):647 - 652 Vol.2.
6. Rodriguez, J., Jih-Sheng Lai, Fang Zheng Peng, "Multilevel inverters: a survey of topologies, controls, and applications" , Industrial Electronics, IEEE Transactions on Volume 49, Issue 4, Aug. 2002 Page(s):724 – 738.
7. De Doncker, R.W., "Twenty years of digital signal processing in power electronics and drives", Industrial Electronics Society, 2003. IECON '03. The 29th Annual Conference of the IEEE, Volume 1, 2-6 Nov. 2003 Page(s):957 - 960 vol.1.

8. Frantz, G., "Digital signal processor trends," *Micro, IEEE* , vol.20, no.6pp.52-59, Nov/Dec 2000.
9. Siriroj Sirisukprasert, “The Modeling and Control of a Cascaded-Multilevel Converter-Based STATCOM”, PhD Dissertation, 2004, Virginia Tech.
10. S. Sirisukprasert, A. Q. Huang, and J. S. Lai, “Modeling, analysis and control of cascaded-multilevel converter-based STATCOM,” in Proc. IEEE-PES general meeting, 2003, vol. 4, July 2003, pp. 13–17.
11. Robert W. Erickson and Dragan Maksimovic, Fundamentals of Power Electronics, 2nd edition, Kluwer Academic Publisher, 2000.
12. B. Zhang, A. Q. Huang, Y. Liu, and S. Atcitty, "Performance of the new generation emitter turn-off (ETO) thyristor," in Proc. IEEE-IAS, 2002, pp. 559-563.
13. S. Sirisukprasert, Z. Xu, B. Zhang, J. S. Lai, and A. Q. Huang, “A high-frequency 1.5 MVA H-bridge building block for cascaded multilevel converters using emitter turn-off thyristor,” in Proc. IEEE-APEC, 2002, pp. 27–32.
14. S. Sirisukprasert, Liu Y., Xu, Z., Zhang B., Zhou X., Hawley, J., Huang A.Q., “Power stage and control design for the ETO-based cascaded-multilevel converter for FACTS applications”, Power Electronics and Motion Control Conference, 2004. IPEMC 2004. The 4th International Volume 3, 14-16 Aug. 2004 Page(s):1111 - 1117 Vol.3.
15. Han C., Yang Z., Chen B., Song W., Huang A.Q., Edris A.-A., Ingram M., Atcitty S., “System integration and demonstration of a 4.5 MVA STATCOM based on emitter turn-off (ETO) thyristor and cascade multilevel converter” , Industrial Electronics Society, 2005. IECON 2005. 32nd Annual Conference of IEEE 6-10 Nov. 2005 Page(s):6 pp.

16. Texas Instrument, “TMS320c6713 DSK Technical Reference”, available from Texas Instrument website.
17. AED-106 manual, available from Signalware corp.
18. Texas Instrument, “TMS320c6000 Technical Brief”, available from Texas Instrument website
19. Texas Instrument, “TMS320C6000 DSP Software-Programmable Phase-Locked Loop (PLL) Controller Reference Guide”, available from Texas Instrument website
20. G. C. Hsieh and J. C. Hung, “Phase-locked loop techniques-A survey,”IEEE Trans. Ind. Electron., vol. 43, pp. 609–615, Dec. 1996.
21. Amuda, L.N., Cardoso Filho, B.J., Silva, S.M., Silva, S.R., Diniz, A.S.A.C., “Wide bandwidth single and three-phase PLL structures for grid-tied PV systems”, Photovoltaic Specialists Conference, 2000. Conference Record of the Twenty-Eighth IEEE 15-22 Sept. 2000 Page(s):1660 – 1663
22. Wei Liu, “Distributed modular controller for high power converter applications”, Master Dissertation, 2005, NCSU
23. Chiasson, J.N., Tolbert, L.M., McKenzie, K.J., Zhong Du, “A New Approach to Solving the Harmonic Elimination Equations for a Multilevel Converter”, Industry Applications Conference, 2003. 38th IAS Annual Meeting. Conference Record of the Volume 1, 12-16 Oct. 2003 Page(s):640 - 647 vol.1
24. Yu Liu, Zhong Du, Alex Q. Huang, Subhashish Bhattacharya, “An Optimal Combination Modulation Strategy for a Seven-level Cascade Multilevel Converter Based STATCOM”, Industry Applications Conference, 2006.

25. Fang Zheng Peng, Jih-Sheng Lai, McKeever, J.W., VanCoevering, J., “A multilevel voltage-source inverter with separate DC sources for static VAr generation”, Industry Applications, IEEE Transactions on Volume 32, Issue 5, Sept.-Oct. 1996 Page(s):1130 - 1138

APPENDICES

Part 1: DSP C code Header file.

1. Header file “tva_consts.h” for constants and memory interface

```
#ifndef __TVA_CONSTS_INCL__
#define __TVA_CONSTS_INCL__
//Math constants
static const float g_cfPi    = (float)(3.14159265);  //Pi
static const float g_cf2Pi  = (float)(6.28318531);  //2*Pi
static const float g_cfPi16          = (float)(0.52359878);  //Pi/6
static const float g_cfPi23  = (float)(2.09439510);  //2/3*Pi
static const float g_cfpi12  = (float)(1.570796327); // pi/2
static const float g_cfpi32  = (float)(4.712388980); // 3*pi/2
static const float g_cfSqt2  = (float)(1.41421356);  //sqrt(2)
static const float g_cfSqt12 = (float)(0.70710678);        //1/sqrt(2)
static const float g_cfSqt23 = (float)(0.81649658);  //sqrt(2/3)
static const float g_cfSqt32 = (float)(0.86602540);  //sqrt(3)/2
static const float g_cfSqt13 = (float)(0.57735027);  //sqrt(1/3)
static const float g_cf23    = (float)(0.66666667);  //2/3
//Constant for phase lock loop
static const float Tpll=5e-5; //20kHz, Calculation frequency of PLL loop,
static const float fKi_pll_om=1;
static const float fKp_pll_om=0;
static const float fKi_pll_vq=1394.59;
static const float fKp_pll_vq=6.28;
//Constant for feedback loop
static const float Ts=10e-5;  //10kHz, Calculation frequency of feedback loop
static const float Ts_IDC=1e-3;   //1kHz
//D-voltage loop
static const float fKi_vd=5.0;//Integral constant for voltage loop
static const float fKp_vd=1.0;//Proportional constant for voltage loop
static const float fKanti_vd=0.0; //Antiwindup constant
static const float flolimit_vd=-3.0; //Upper limit
static const float fuplimit_vd=3.0;         //Lower limit

// Individual DC loop
static const float fKi_IDC=0.005;  //Integral constant for individual voltage loop
static const float fKp_IDC=0.0005; //Proportional constant for individual voltage loop
static const float fKanti_IDC=0.0; //Antiwindup constant for D-channel current loop
static const float flolimit_IDC=-0.2865;  //Upper limit
static const float fuplimit_IDC=0.2865;   //Lower limit

//D-current loop
static const float fKi_id=1.3;   //Integral constant for D-channel current loop
```

```
    static const float fKanti_id=0.0; //Antiwindup constant for D-channel current loop
    static const float flolimit_id=-1.4; //Upper limit
    static const float fuplimit_id=1.4;          //Lower limit

    //Q-current loop
    static const float fKi_iq=1.3;    //Integral constant
    static const float fKp_iq=0.013;  //Proportional constant
    static const float fKanti_iq=0.0; //Antiwindup constant for D-channel current loop
    static const float flolimit_iq=-1.4; //Upper limit -1.098
    static const float fuplimit_iq=1.4;          //Lower limit

    //For FPGA on the daugther board
    //ADC Addresses
    static unsigned int* volatile pADCTL = (unsigned int*)(0xA0210400);

    //ADC1 (U5)
    static const unsigned int* volatile padc1 = (unsigned int*)(0xA0210800);
    static const unsigned int* volatile padc2 = (unsigned int*)(0xA0210C00);
    static const unsigned int* volatile padc3 = (unsigned int*)(0xA0211000);
    static const unsigned int* volatile padc4 = (unsigned int*)(0xA0211400);
    //ADC2 (U6)
    static const unsigned int* volatile padc5 = (unsigned int*)(0xA0211800);
    static const unsigned int* volatile padc6 = (unsigned int*)(0xA0211C00);
    static const unsigned int* volatile padc7 = (unsigned int*)(0xA0212000);
    static const unsigned int* volatile padc8 = (unsigned int*)(0xA0212400);
    //ADC3 (U7)
    static const unsigned int* volatile padc9 = (unsigned int*)(0xA0212800);  //address of
Io A
    static const unsigned int* volatile padc10 = (unsigned int*)(0xA0212C00); //address of
Io B
    static const unsigned int* volatile padc11 = (unsigned int*)(0xA0213000); //address of
Io C
    static const unsigned int* volatile padc12 = (unsigned int*)(0xA0213400); //address of
vpccABC.AB
    //ADC4 (U8)
    static const unsigned int* volatile padc13 = (unsigned int*)(0xA0213800);
    static const unsigned int* volatile padc14 = (unsigned int*)(0xA0213C00);
    static const unsigned int* volatile padc15 = (unsigned int*)(0xA0214000); //address of
vpccABC.BC
    static const unsigned int* volatile padc16 = (unsigned int*)(0xA0214400); //address of
Vdc
    //PWM register addresses
    static unsigned int* volatile SW_state_A    = (unsigned int*)(0xA0214C00);
    static unsigned int* volatile SW_state_B    = (unsigned int*)(0xA0214800);
```

```
static unsigned int* volatile SW_state_C    = (unsigned int*)(0xA0215000);
static unsigned int* volatile pPWMA_CONTROL  = (unsigned int*)(0xA0215400);
//Digital output address
static unsigned int* const pDIGIOin = (unsigned int*)(0xA0215800);
static unsigned int* volatile HMI_com2  = (unsigned int*)(0xA0216400);  //only use com2
static unsigned int* volatile HMI_com1  = (unsigned int*)(0xA0216C00);  //com 1 don't work
#endif
```

2. Header file “tva_control.h”

```
//Header file for control parameter inputs

#ifndef __TVA_CONTROL_INCL__
#define __TVA_CONTROL_INCL__
#include "tva_types.h"
static const float MaxDC=100.0;                    //Allowable maximum DC bus voltage
static const float MinDC=-20.0;          //Allowable minimum DC bus voltage
static const float pt05_vd_ref=0.05*100; //5% of dc bus setting
static const float bus_ref=58.3;          //DC bus references: 58.3
static const float Vdc_dio=32.0;          //DC voltage after Diode charge = Vpcc_L-L_peak/2 (3 level)
static const float Vdc_discharge=40.0;   //DC voltage after discharge
//Output current param
static const float MaxI=30.0;             //Allowable maximum peak output current %need to be update after check the new resistor
static const float MinI=-30.0;     //Allowable minimum peak output current      %need to be update after check the new resistor
static const float Max_norm_Iout = 4.0;          //Max_norm_Iout = Max_norm_Iout in abc*sqrt(2)*1.22
TVA_DQO_DATA park( const TVA_ABC_DATA* data,const float cf_cos, const float cf_sin);
TVA_ABC_DATA inv_park( const TVA_DQO_DATA* data,const float cf_cos, const float cf_sin);
void UpdatePI( PI_PARAMS* prPI );
void UpdatePIAW( PIAW_PARAMS* prPI);
void statcom(const TVA_DQO_DATA duty, const float delta_t, STATCOM_PARAM* prSTATCOM);
void PLL();
void phase_lock_loop(TVA_ABC_DATA* abc_param);
void Update_param();
static SW_forInverter Optimal_PWM (const TVA_DQO_DATA );
```

```
static SW_forPhase Phase_PWM (const float M, const float angle_shift);
void TVA_intr_setup (int num, int isn,void(*ptr_isr)(void));
void TVA_timer_setup (int num, unsigned int period);
#endif
```

3. Header file “tva_types.h” . This file defines all the data types, structures for the code.

```
//Header file for the data type, structures and unions
#ifndef __TVA_TYPES_INCLUDE__
#define __TVA_TYPES_INCLUDE__

typedef struct  _TVA_DQO_DATA{
     float D;   //Direct
     float Q;   //Quadrature
     float O;   //Zero component (specified as the letter "oh")
} TVA_DQO_DATA;
typedef struct  _TVA_ABC_DATA{
    volatile float A;   //Phase A data
    volatile float B;   //Phase B data
    volatile float C;   //Phase C data
} TVA_ABC_DATA;
typedef struct  _TVA_INT_ABC_DATA{
    unsigned int A;   //Phase A data
    unsigned int B;   //Phase B data
    unsigned int C;   //Phase C data
} TVA_INT_ABC_DATA;
typedef struct  _TVA_BUFF{
    float od;   //Buffer for old number
    float nw;   //Buffer for new number
} TVA_BUFF;
typedef struct  _ANG{
    float cos;   //Cosine term of the angle information
    float sin;   //Sine term of the angle informaton
} ANG;
//Struct for PI controller parameters
typedef struct _PI_PARAMS{
    float fTs;                  //sampling rate
    float fKe_1;                //constant of Ki*Ts/2+Kp
    float fKe;                  //constant of Ki*Ts/2-Kp
    float fErrPrev;             //buffer for Prev Error term on input
    float fStatePrev;           //state for integrator
    float* pfPosInput;  //pointer to positive input (Refference)
    float* pfNegInput;  //pointer to negative input (Feedback)
    float fOutput;      //output result
```

```
} PI_PARAMS;
//Struct fot PI with antiwindup parameters
typedef struct _PIAW_PARAMS{
    float fTs;              //sampling rate
    float fKii;             //constant of Ki*Ts/2
    float fKp;              //constant of Kp
    float fKanti;           //constant of antiwindup
    float uplimit;          //upper limit
    float lolimit;          //lower limit
    float fErrPrev;         //buffer for Prev Error term on input
    float fStatePrev;       //state for integrator
    float* pfPosInput;  //pointer to positive input (Refference)
    float* pfNegInput;  //pointer to negative input (Feedback)
    float fOutput;      //output result
} PIAW_PARAMS;
//Struct for STATCOM model parameters
typedef struct _STATCOM_PARAM{
    float id;                   //D-channel current
    float iq;                   //Q-channel current
    float e;                //DC bus voltage
} STATCOM_PARAM;

// Struct for STATCOM model in ABC coordinate
typedef struct _STATCOMABC_PARAM{
    float ia;                   //D-channel current
    float ib;                   //Q-channel current
    float ic;
    float e;                //DC bus voltage
} STATCOMABC_PARAM;
typedef struct _LPF_PARAMS{
   float preInput;    //input(k-1)
   float Input;       //input(k)
   float Output;       // output(k)
   float t1;
   float t2;
   float t3;
} LPF_PARAMS;
typedef struct _SW_forHBridge{
    short L1;               //A  H-bridge SW 1st and 3rd switching signals
    short L2;               //A  H-bridge SW 2nd and 4th switching signals
} SW_forHBridge;

typedef struct _SW_forPhase{
    SW_forHBridge H1;   // switching signals for the first H-bridge in a phase
```

```
} SW_forPhase;

typedef struct _SW_forInverter{
    SW_forPhase A;              // switching signals for the A phase
    SW_forPhase B;              // switching signals for the B phase
    SW_forPhase C;              // switching signals for the C phase
} SW_forInverter;

typedef struct _ang_Hbridge{
    unsigned short ag0;
    unsigned short ag1;
    unsigned short ag2;
    unsigned short ag3;
    unsigned short ag4;
} ang_Hbridge;
#endif //__TVA_TYPES_INCLUDE__
```

Part 2: DSP C source code.

1. Main Function “RTDXTEST.c”

```
// Main function
// For the procedure of communications among interrupt, timer and RTDX
#include <stdlib.h>
#include <stdio.h>
#include <std.h>
#include <rtdx.h>
#include <math.h>
#include "RTDXTESTcfg.h"
#include "timer.h"
#include "intr.h"
#include "regs.h"
#include <csl.h>
#include "tva_consts.h"
// RTDX header file
/* TARGET_INITIALIZE()*/
#include "target.h"
#define pi 3.1415927
// count index
short  statcom_status = 1;
short  button=0;
short  vdc_status = 0;
short  RTDX_output[5];
short  RTDX_input[2];
```

```
extern float E;
extern float id;
extern float iq;
extern int int_bus_status;
short  dcbus_A=0;
short  dcbus_B=0;
short  dcbus_C=0;
float Iqref = 0.0;
int fault_ct=0;
unsigned short gct=0;
unsigned short psct=99;
short  vdcindicator=0;
short  appl_err = 0;
void HWI_init(void);
void applicationISR(void);
void main()
{
   CSL_cfgInit();
   DSK6713_waitusec(10000);
   DSK6713_init();
   DSK6713_waitusec(10000);
    HWI_init();
   DSK6713_waitusec(5000);
    appl_init();     //Initailization for the feedback loop
   DSK6713_waitusec(10000);
  /* Target Specific Initialization  */
  TARGET_INITIALIZE();
  RTDX_output[0]=1;// initialize statcomstatus
  RTDX_output[1]=1;// initialize vdc_indicator
  /* Enable the RTDX channels */
   RTDX_enableOutput(&ochan);
   RTDX_enableInput(&ichan);
    TIMER_START(0);
}

void applicationISR(void)
{

    if (appl_err != 0)
    {
        *pDIGIOin = 0x002;
        //disable all switching signals
        *SW_state_A = 0x3F;
        *SW_state_C = 0x0;
```

```
fault_ct++;
        switch (appl_err)
        {
                case 2:
                //"Fault!!! OVER DC BUS VOLTAGE"
                statcom_status=7;
                RTDX_output[0]=7;
                if (fault_ct >5)
            {
                  RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                  fault_ct=0;
            }
                break;
                case 3:
                //"Fault!!! OVER OUTPUT CURRENT"
                statcom_status=8;
                RTDX_output[0]=8;
                if (fault_ct >1)
            {
                  RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                  fault_ct=0;
            }
                break;
                case 4:
                //"Fault!!! AC VOLTAGE OUT OF RANGE"
                statcom_status=9;
                RTDX_output[0]=9;
                if (fault_ct >1)
            {
                  RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                  fault_ct=0;
            }
                break;
                case 5:
                //"Successful discharge to 400V and automatically shut down"
                statcom_status=10;
                RTDX_output[0]=10;
                if (fault_ct >1)
            {
                  RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));
```

```
fault_ct=0;
            }
                break;
                case 6:
                //"Manually shut down the system"
                statcom_status=11;
                RTDX_output[0]=11;
                if (fault_ct >1)
            {
                 RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                 fault_ct=0;
            }
                break;
                case 7:
                //"Fault!!! PLL fault!!"
                statcom_status=12;
                RTDX_output[0]=12;
                if (fault_ct >1)
            {
                 RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                 fault_ct=0;
            }
                case 80:
                //"Combined Fault!!! Check SW signals"
                statcom_status=13;
                RTDX_output[0]=13;
                if (fault_ct >1)
            {
                 RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                 fault_ct=0;
            }
            break;
                default:
                //"Fault!!! Don't know"
                statcom_status=14;
                RTDX_output[0]=14;
                if (fault_ct >1)
            {
                 RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));

                 fault_ct=0;
```

```
}
            }

   }
   else
    {
     // If without fault detected, ISR will call application function
      appl_err= appl_process();
     RTDX_output[0]= statcom_status;
     RTDX_output[1]= int_bus_status;
     RTDX_output[2]=E;
     RTDX_output[3]=id;
     RTDX_output[4]=iq;
     Iqref = RTDX_input[0];
     button= RTDX_input[1];
      gct++;
      psct++;
      if (RTDX_output[0]==0) vdcindicator++;
      if (gct>699)
       {
         gct=0;
         if (!RTDX_channelBusy(&ichan))
         {
           RTDX_readNB(&ichan, &RTDX_input, sizeof(RTDX_input));
         }
       }
     if ((!RTDX_channelBusy(&ichan))&&(gct!=0))
     {
      if ( psct>799)
            {
              psct=0;
              RTDX_write( &ochan, &RTDX_output, sizeof(RTDX_output));
            }
     }
    }
}
//Initialize interrupts
void HWI_init(void)
{
   IRQ_globalEnable();
   IRQ_enable(IRQ_EVT_TINT0);
}
```

2. Interrupt Service Routine Application Function. “Aed_106_TVA.c”

```
// ISR application function
//+++TVA included hearders
#include "tva_types.h"
#include "tva_control.h"
#include "tva_consts.h"
#include "three_lev_angletable.h"
#include "asintable.h"
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <timer.h>
// +++++ TVA global variables
#define nGraphDataSize 1000
#define nGraphDataSize1 1000
#define aver_size 10
//extern int appl_err;
extern short int button;
extern short int statcom_status;
extern short int vdc_status;
extern float Iqref;
float dc_A;
float dc_B;
float dc_C;
float E=0.0;
float id=0.0;
float iq=0.0;
float ag =0.0;
float dt=0.0;
short tz=0;
// communication protocol variables:
unsigned int frame_start=0;
unsigned int frame_stop=0;
unsigned int data1=0; // First H-bridge DC voltage
unsigned int data2=0; // First H-bridge H-bridge current
unsigned int data3=0; // Second H-bridge DC voltage
unsigned int data4=0; // Second H-bridge Current
unsigned int data5=0; // Third H-bridge DC voltage
unsigned int data6=0; // Third H-bridge current

//Counters
unsigned int g_nPBCtr = 0;       //Counter for push button loop
unsigned int g_nFBCtr = 2;       //Counter for feedback loop
unsigned int g_nDCCtr = 0;       //Counter for DC bus checking
unsigned int g_nLoopCtr = 0;//counter for graph
```

```
//Graph data
float VA[nGraphDataSize];
float VB[nGraphDataSize];
float VC[nGraphDataSize];
float Ia[nGraphDataSize];
float Ib[nGraphDataSize];
float vpcc_fault[200];
int g_nDioCharge=0;    //counter for graph of diode rectifier charge
int g_nPWMCharge=0;    //counter for graph of boost rectifier PWM charge
int g_nService=0;      //counter for graph of online service mode
int g_nDischarge=0;    //counter for graph of discharge mode
long t=0;
int tt=0;
int ttt=0;
int faultct=0;
int g_nVdcAVG = 0; // counter to calculate the mean DC voltage.
int DCloopcount=0;  //DC voltage loop counter
static volatile short int DCcounta=0;
static volatile short int DCcountb=3;
static volatile short int DCcountc=6;
short int DCstat=0;
short int av=0;
float aver_A[aver_size];
float aver_B[aver_size];
float aver_C[aver_size];
const float vpcc_OV=900.0;
float Vdc_pwmc=0.0;
short delta_t=0; // time counter for vpcc simulation

//--------------------Measurement Parameters-----------------------------
TVA_ABC_DATA vpccABC;              //Voltage at the point of common coupling
TVA_ABC_DATA dcbus;                //DC bus voltages
TVA_ABC_DATA ioutABC;              //Converter output currents
TVA_ABC_DATA DC_temp;
TVA_ABC_DATA DC_average;    // average DC bus voltage
TVA_ABC_DATA Isim;                 //For simulation
TVA_ABC_DATA sim_const;
TVA_ABC_DATA theta_temp; // used to generate duty simulation output
TVA_ABC_DATA theta_temp2;
TVA_ABC_DATA theta_temp3;
TVA_DQO_DATA IDQO;         //for test
STATCOMABC_PARAM sim;       // For statcom simulation in ABC
TVA_ABC_DATA vpccsim;
//--------------------STATCOM Parameters-----------------------------------------
```

```
    STATCOM_PARAM statcom_pr;   //Output of STATCOM model

//--------------------Parameter for PLL loop-----------------------------------
    PI_PARAMS PI_Vpccq;        //Struture of PI compensator of PLL loop
    PI_PARAMS PI_Omega;        //Struture of PI compensator of PLL loop
   LPF_PARAMS LPF_dc;  // LPF parameters for Vdc
   LPF_PARAMS LPF_dcA; // LPF parameters for Vdc
   LPF_PARAMS LPF_dcB; // LPF parameters for Vdc
   LPF_PARAMS LPF_dcC; // LPF parameters for Vdc
   LPF_PARAMS LPF_id;  // LPF parameters for id
   LPF_PARAMS LPF_iq;  // LPF parameters for iq

    float ftheta=0;           //Theta of line voltage (0 to 2pi)
    float cf_cos = 1;         //Cosin from Sine table
    float cf_sin = 0;         //Sin from Sine table
    float delta;              //Test phase shift of line voltages
    float ftheta_ph=0;        //Theta of phase voltage (0 to 2pi)
    float cf_cos_ph = 1.0;          //Cosin from Sine table
    float cf_sin_ph = 0.0;          //Sin from Sine table
    float theta_err = 0.0;      //error of PLL output
    float theta_prev =0.0;      //record of previous theta value
     int theta_ct=0;           // time couner,
   int thetaerr_ct=0;         //counter to record the theta error

   volatile unsigned int timer_ct=0;
   volatile unsigned int timer_ct2=0;
   volatile int tick=0;
//--------------------Parameter for OPWM-------------------
    static volatile SW_forInverter Inverter_out;
    float M_test = 0.0;
//--------------------Parameter for Feedback Loop------------
    //For Vdc loop
    float dcbus_average=31.0; //Average value of three DC bus voltages
    float dcbus_average_prev1=31.0; //For debug
    float dcbus_average_prev2=31.0; //For debug
    float dcbus_average_prev3=31.0;

    TVA_DQO_DATA ioutDQO; //Converter output currents in DQO
    TVA_DQO_DATA dutyDQO;//Duty cycle in DQO
    TVA_ABC_DATA dutyABC;  //Duty cycle in ABC

    float vd_ref=0;                    //DC Bus setting
    float vd_fb=0;                 //DC Bus feedback
    PI_PARAMS PI_VD;           //Structure of PI compensator for Vd loop
```

```
    //For Id loop
    float id_ref=0;             //D-channel current reference = output of Vd-loop
    float id_fb=0;              //D-channel current feedback
    PI_PARAMS PI_ID;            //Structure of PI compensator for Id loop

    //For Iq loop
    float iq_ref=0;             //Q-channel current reference = Iq command
    float iq_fb=0;              //Q-channel current feedback
    PI_PARAMS PI_IQ;            //Structure of PI compensator for Iq loop

    //For debugging
    PIAW_PARAMS PIT_ID;             //Test PI with antiwindup

    PIAW_PARAMS PIT_VD;

    PIAW_PARAMS PIT_IQ;

//--------------------Parameter for Individual DC loop-------
    PIAW_PARAMS PI_DC_A;
    PIAW_PARAMS PI_DC_B;
   PIAW_PARAMS PI_DC_C;
   float DC_A = 0.0;
   float DC_B = 0.0;
   float DC_C = 0.0;
   float delta_A = 0.0;
   float delta_B = 0.0;
   float delta_C = 0.0;

//--------------------End of Parameter for Feedback Loop---------

//-------------ADC Parameter---------------------------------
    float fadc1 = 0.0;      //ADC 1 to 16
  static  volatile float fadc2 = 0.0;
  static  volatile float fadc3 = 0.0;
  static  volatile float fadc4 = 0.0;
    float fadc5 = 0.0;
    float fadc6 = 0.0;
    float fadc7 = 0.0;
    float fadc8 = 0.0;
    float fadc9 = 0.0;
    float fadc10 = 0.0;
    float fadc11 = 0.0;
    float fadc12 = 0.0;
    float fadc13 = 0.0;
```

```
   float fadc14 = 0.0;
   float fadc15 = 0.0;
   float fadc16 = 0.0;

   float delta_dc = 0.0;
   static volatile float dcA_prev = 31.0;
   static volatile float dcB_prev = 31.0;
   static volatile float dcC_prev = 31.0;
   static volatile float M = 1.3;
//---------End of ADC Parameter----------------------------

    int i=0;                                    //A test counter
    int j=0;                                    //A counter for plot
    int mode=0;                                 //Operation mode index
    int charge=0;
    int run=0;
    int stop=0;
    int ac_disconnect=0;
    int discharge=0;
   int       emergency_stop=0;
   int combine_fault=0;
   int initialize=0;

   int CVdca=0;          // counter for DC voltage of phase A
    int CVdcb=0;          // counter for DC voltage of phase B
    int CVdcc=0;          // counter for DC voltage of phase C
    int CIa=0;           // counter for current of phase A
    int CIb=0;           // counter for current of phase B
    int CIc=0;           // counter for current of phase C
    int CVab=0;           // counter for  Vab
    int CVbc=0;           // counter for  Vbc
//  const int safeV=4;
    int int_bus_status=1;  //DC bus ready flag
                                                    // 1 for fully charged by retifier
    int bus_ready=0;                // 1 is ready to normally operate
//+++ End of TVA global variables

/*****************************************************************
  switch_period_int - interrupt service routine for switching cycle
  interupt.
*****************************************************************/

//Declarate operation modes
void appl_init(void);                       //Mode 1: Initialization
```

```
int TVA_idle();                                  //Mode 2: Idle
int TVA_charge();                                //Mode 3: Charge
int TVA_run();                                   //Mode 4: Run
int TVA_stop();                                  //Mode 5: Stop
int TVA_AC_disconnect();                 //Mode 6: AC Circuit Brakeritch disconnect
int TVA_discharge();                     //Mode 7: Discharge
static SW_forInverter Optimal_PWM ();
//static SW_forPhase Phase_PWM ();
static SW_forPhase Phase_PWM (const float, const float);
//Declarate PLL function
void PLL();

//-----------------------------------------
/*************************************************************************
  appl_init - Performs the following tasks:
    - Assign mode status to 1:initialization
    - Disable all switching blocks

*************************************************************************/
void appl_init(void)
{
    *pDIGIOin = 0x004;
    *SW_state_A = 0x3F; // all switch off
    *SW_state_C = 0x0;


// for debug purpose
    statcom_pr.id=0;
    statcom_pr.iq=0;
    statcom_pr.e=33.0;


    dutyDQO.D=1.4;
    dutyDQO.Q=0;
    dutyDQO.O=0;

    DC_average.A=0.0;
    DC_average.B=0.0;
    DC_average.C=0.0;

    DC_temp.A=0.0;
    DC_temp.B=0.0;
    DC_temp.C=0.0;
```

```
//----------Initial PLL---------------------------
    //Vq loop
    PI_Vpccq.fTs = Tpll;
    //PLL sampling rate
    PI_Vpccq.fKe = fKi_pll_vq*PI_Vpccq.fTs/2+fKp_pll_vq;   //Constant of Ki*Ts/2+Kp
    PI_Vpccq.fKe_1 = fKi_pll_vq*PI_Vpccq.fTs/2-fKp_pll_vq; //Constant of Ki*Ts/2-Kp

    PI_Vpccq.fErrPrev = 0;                  //Buffer for Prev Error term on input
    PI_Vpccq.fStatePrev = 0;            //State for integrator


    //Omega loop
    PI_Omega.fTs = Tpll;
    //PLL sampling rate
    PI_Omega.fKe = fKi_pll_om*PI_Omega.fTs/2+fKp_pll_om;        //Constant of
Ki*Ts/2+Kp
    PI_Omega.fKe_1 = fKi_pll_om*PI_Omega.fTs/2-fKp_pll_om;      //Constant of
Ki*Ts/2-Kp

    PI_Omega.fErrPrev = 0;                  //Buffer for Prev Error term on input
    PI_Omega.fStatePrev = 0;            //State for integrator


//----------Initial Feedback Loop-----------------

//Id loop
    PIT_ID.fTs = Ts;
    PIT_ID.fKii = Ts*fKi_id;
    PIT_ID.fKp = fKp_id;
    PIT_ID.lolimit = -1.4;
    PIT_ID.uplimit = 1.4;
    PIT_ID.fKanti = fKanti_id*PIT_ID.fKii;
    PIT_ID.fErrPrev = 0.0;                  //Buffer for Prev Error term on input
    PIT_ID.fStatePrev = 1.4;//0.7*1.22;//1.39;          //State for integrator
    PIT_ID.fOutput = 1.4;//0.7*1.22;//1.39;

//Vdc loop
    PIT_VD.fTs = Ts;
    PIT_VD.fKii = Ts*fKi_vd;
    PIT_VD.fKp = fKp_vd;
    PIT_VD.lolimit = -5.0;   // maximum output current RMS 5A,
    PIT_VD.uplimit = 5.0;
    PIT_VD.fKanti = fKanti_vd*PIT_VD.fKii;
    PIT_VD.fErrPrev = 0.0;              //Buffer for Prev Error term on input
```

```
    PIT_VD.fStatePrev = 0.0;                //State for integrator
    PIT_VD.fOutput = 0.0;
    vd_ref = bus_ref;                       //DC Bus setting
    Vdc_pwmc=1.0*bus_ref; //0.991*bus_ref;      //99.9%*1200V of DC bus voltage

// Individual DC loop A

    PI_DC_A.fTs = Ts_IDC;         // PI loop frequency is 1kHz
    PI_DC_A.fKii = Ts_IDC*fKi_IDC;
    PI_DC_A.fKp = fKp_IDC;
    PI_DC_A.lolimit = flolimit_IDC;   // maximum output current RMS 5A,
    PI_DC_A.uplimit = fuplimit_IDC;
    PI_DC_A.fKanti = 0.0;
    PI_DC_A.fErrPrev = 0.0;                     //Buffer for Prev Error term on input
    PI_DC_A.fStatePrev = 0.0;               //State for integrator
    PI_DC_A.fOutput = 0.0;

 // Individual DC loop B

    PI_DC_B.fTs = Ts_IDC;         // PI loop frequency is 1kHz
    PI_DC_B.fKii = Ts_IDC*fKi_IDC;
    PI_DC_B.fKp = fKp_IDC;
    PI_DC_B.lolimit = flolimit_IDC;   // maximum output current RMS 5A,
    PI_DC_B.uplimit = fuplimit_IDC;
    PI_DC_B.fKanti = 0.0;
    PI_DC_B.fErrPrev = 0.0;                     //Buffer for Prev Error term on input
    PI_DC_B.fStatePrev = 0.0;               //State for integrator
    PI_DC_B.fOutput = 0.0;

// Individual DC loop C

    PI_DC_C.fTs = Ts_IDC;         // PI loop frequency is 1kHz
    PI_DC_C.fKii = Ts_IDC*fKi_IDC;
    PI_DC_C.fKp = fKp_IDC;
    PI_DC_C.lolimit = flolimit_IDC;   // maximum output current RMS 5A
    PI_DC_C.fErrPrev = 0.0;                     //Buffer for Prev Error term on input
    PI_DC_C.fStatePrev = 0.0;               //State for integrator
    PI_DC_C.fOutput = 0.0;

//Iq loop
    PIT_IQ.fTs = Ts;
    PIT_IQ.fKii = Ts*fKi_iq;
    PIT_IQ.fKp = fKp_iq;
    PIT_IQ.lolimit = -1.4;
```

```
PIT_IQ.uplimit = 1.4;
 PIT_IQ.fKanti = fKanti_iq*PIT_IQ.fKii;
 PIT_IQ.fErrPrev = 0.0;                          //Buffer for Prev Error term on input
 PIT_IQ.fStatePrev = 0.0;                //State for integrator
 PIT_IQ.fOutput = 0.0;
LPF_dc.preInput=0.0;
LPF_dc.Output=0.0;
LPF_dc.t1=0.98019;    // Vdc 200Hz LPF
LPF_dc.t2=0.0099056;
LPF_dc.t3=0.0099056;

LPF_dcA.preInput=0.0;
LPF_dcA.Output=0.0;
LPF_dcA.t1=0.98019;    // Vdc 200Hz LPF
LPF_dcA.t2=0.0099056;
LPF_dcA.t3=0.0099056;
LPF_dcB.preInput=0.0;
LPF_dcB.Output=0.0;
LPF_dcB.t1=0.98019;    // Vdc 200Hz LPF
LPF_dcB.t2=0.0099056;
LPF_dcB.t3=0.0099056;

LPF_dcC.preInput=0.0;
LPF_dcC.Output=0.0;
LPF_dcC.t1=0.98019;    // Vdc 200Hz LPF
LPF_dcC.t2=0.0099056;
LPF_dcC.t3=0.0099056;


LPF_id.preInput=0.0;
LPF_id.Output=0.0;
LPF_id.t1=0.98019;    // Id 1200Hz LPF
LPF_id.t2=0.0099056;
LPF_id.t3=0.0099056;

LPF_iq.preInput=0.0;
LPF_iq.Output=0.0;
LPF_iq.t1=0.98019;    // Iq 1200Hz LPF
LPF_iq.t2=0.0099056;
LPF_iq.t3=0.0099056;
 dcbus.A=31.0;
dcbus.B=31.0;
dcbus.C=31.0;
```

```
    ioutABC.A=0.0;
    ioutABC.B=0.0;
    ioutABC.C=0.0;

    ioutDQO.D=0.0;
    ioutDQO.Q=0.0;
    ioutDQO.O=0.0;

    vpccABC.A=0.0;
    vpccABC.B=0.0;
    vpccABC.C=0.0;

    dcbus.A = 31.0;
    dcbus.B = 31.0;
    dcbus.C = 31.0;

    DC_average.A = 33.0;
    DC_average.B = 33.0;
    DC_average.C = 33.0;
    // setup initial value for the aver array
    for (av=0;av<aver_size;av++)
    {
      aver_A[av]=33.0;
      aver_B[av]=33.0;
      aver_C[av]=33.0;

    }
    puts("TVA-DSTATCOM controller is running.");
     puts("Control system has been initialized.");

} /* end appl_init */


/*****************************************************************
  appl_process - Processes the following tasks:
    - Reading user push buttons
    - Multiple input protection
    - Jump to the corresponding mode

*****************************************************************/
int appl_process(void)
{
```

```
    int ret=0;
    int pb_status=0;                    //Push button status


    vdc_status = int_bus_status;
    g_nPBCtr++;                                 //Counter for push button cycle

//Read Voltage @ point of common coupling

    vpccABC.A = (1989.0f-*padc12)*0.310095f;
    vpccABC.B = (1982.0f-*padc15)*0.296302f;
    vpccABC.C = -(vpccABC.A+vpccABC.B);
    dcbus_average=(dcbus.A+dcbus.B+dcbus.C)/3;
     dcbus_average=(dcbus_average+dcbus_average_prev1
     +dcbus_average_prev2+dcbus_average_prev3)/4;
     dcbus_average_prev3=dcbus_average_prev2;
     dcbus_average_prev2=dcbus_average_prev1;
     dcbus_average_prev1=dcbus_average;

//read Series communcation data from FPGA
                    data1= (*padc2 & 0x001FF000)>>12;
                    data2= (*padc3 & 0x001FF000)>>12;
                    data3= (*padc4 & 0x001FF000)>>12;
                    data4= (*padc5 & 0x001FF000)>>12;
                    data5= (*padc6 & 0x001FF000)>>12;
                    data6= (*padc7 & 0x001FF000)>>12;

          fadc2= 2.77777829f*(data1-125.0f);
          LPF_dcA.Input=fadc2;
          LPF(&LPF_dcA);
          dcbus.A =LPF_dcA.Output;
          fadc3 = 2.77777814f*(data2-125.0f);
          LPF_dcB.Input=fadc3;
          LPF(&LPF_dcB);
          dcbus.B =LPF_dcB.Output;
          fadc4 = 2.78611111f*(data3-119.0f);
          LPF_dcC.Input=fadc4;
          LPF(&LPF_dcC);
          dcbus.C =LPF_dcC.Output;
    if ((dcbus.A > MaxDC)|(dcbus.A < MinDC))
              {
              CVdca++;
              }
```

```
            {

CVdca=0;
            }
            if(CVdca>10)
              {
              ret = 02;
              return ret;
              }
   if ((dcbus.B > MaxDC)|(dcbus.B < MinDC))
            {
            CVdcb++;
            }
            else
            {
            CVdcb=0;
            }
            if(CVdcb>10)
              {
              ret = 02;
              return ret;
              }
   if ((dcbus.C > MaxDC)|(dcbus.C < MinDC))
            {
            CVdcc++;
            }
            else
            {
            CVdcc=0;
            }
            if(CVdcc>10)
              {
              ret = 02;
              return ret;
              }

   ioutABC.A =(*padc9-1956.7f)*0.01522336f;
   ioutABC.B =(*padc10-1993.6f)*0.0155933f;
  ioutABC.C =(*padc11-1936.8f)*0.01020482f;
  if ((ioutABC.A > MaxI)|(ioutABC.A < MinI))
            {
            CIa++;
            }
            else
```

```
{
        CIa=0;
        }

if(CIa>30)
        {
        ret = 03;
        return ret;
        }
    if ((ioutABC.B > MaxI)|(ioutABC.B < MinI))
        {
        CIb++;
        }
        else
        {
        CIb=0;
        }
        if(CIb>30)
        {
        ret = 03;
        return ret;

        }
    if ((ioutABC.C > MaxI)|(ioutABC.C < MinI))
    {
        CIc++;
        }
        else
        {
        CIc=0;
        }
        if(CIc>30)
        {
        ret = 03;
        return ret;
        }

    if ((vpccABC.A > vpcc_OV)|(vpccABC.A < -vpcc_OV))
    {
        CVab++;
        }
        else
        {
        CVab=0;
```

```
}
            if(CVab>2)
            {
            ret = 04;
            return ret;
            }

    if ((vpccABC.B > vpcc_OV)|(vpccABC.B < -vpcc_OV))
    {
            CVbc++;
            }
            else
            {
            CVbc=0;
            }
            if(CVbc>2)
            {
            ret = 04;
            return ret;
            }
// **********************End of ADC reading**************************
// ********************* push buttons reading **********************
//When the voltage excess 0.35V, the button is pushed
            //----CHARGE-----------
            if (button == 2)
            {
                    charge=1;
                    pb_status++;
            }
            //----Initialize--------------
            if (button == 3)
            {
                    initialize=1;
                    pb_status++;
            }
            //else run =0;
            //----discharge---------
            if (button == 4)
            {
                    discharge=1;
                    pb_status++;
            }
            //----ac_disconnect---------
```

```
if (button == 5)
        {
                ac_disconnect=1;
                pb_status++;
        }

        //----STOP-------------
        if (button == 1)
        {
                stop=1;
                pb_status++;
        }

        if (button == 0)
        {
                stop=0;
                pb_status++;
        }
E =dcbus.A;
id=dcbus.B;
iq=dcbus.C;

if (int_bus_status <3) {phase_lock_loop(&vpccABC); }
// initislize AC connection
   if ((initialize == 1)&&(int_bus_status == 1))
    {

      *SW_state_A = 0x3F;
      *SW_state_C = 0x0;
      statcom_status=3;
      ac_disconnect=0;
      initialize=0;
      int_bus_status = 2;

   }
   if ((charge == 1)&&(int_bus_status == 2))
   {

     ac_disconnect=0;
     run=0;
     g_nVdcAVG++;            // counter to calculate the average DC voltge
     ret=0;
     if (DCstat > 150)
```

```
{

        *pDIGIOin = 0x005;
        int_bus_status = 3;       //finish diode mode charge
        g_nPWMCharge = g_nDioCharge;  //contineous to record PWM waveform
        charge = 0; // reset charge button
        }

        ioutDQO = park(&ioutABC,cf_cos_ph,cf_sin_ph);

              ttt++;
        if ((ttt > 4)&&(g_nDioCharge < 1000))
              ttt=0;
              g_nDioCharge++;
              VA[g_nDioCharge]=LPF_dcA.Output;
              VB[g_nDioCharge]=LPF_dcB.Output;
              VC[g_nDioCharge]=LPF_dcC.Output;
              Ia[g_nDioCharge]=vpccABC.B;
              Ib[g_nDioCharge%1000]=vpccABC.A;
              }


        if (g_nVdcAVG>9)     // calculate average DC voltage
         g_nVdcAVG=0;
         DC_temp.A=0;
         DC_temp.B=0;
         DC_temp.C=0;
         for (av=0;av<aver_size-1;av++)
          {
            aver_A[av]=aver_A[av+1];
            aver_B[av]=aver_B[av+1];
            aver_C[av]=aver_B[av+1];
            DC_temp.A=DC_temp.A+aver_A[av];
            DC_temp.B=DC_temp.B+aver_B[av];
            DC_temp.C=DC_temp.C+aver_C[av];
          }
         aver_A[aver_size-1]=dcbus.A;
         aver_B[aver_size-1]=dcbus.B;
         aver_C[aver_size-1]=dcbus.C;
         DC_average.A=(DC_temp.A+ aver_A[aver_size-1])/aver_size;
         DC_average.B=(DC_temp.B+ aver_B[aver_size-1])/aver_size;
         DC_average.C=(DC_temp.C+ aver_C[aver_size-1])/aver_size;
         if ((DC_average.A > Vdc_dio) && (DC_average.B > Vdc_dio) &&
(DC_average.C > Vdc_dio))
```

```
{
                if(DCstat < 151)   // 75ms
                {
                 DCstat++;
                }
               }
               else
               {
                DCstat=0;
               }

     }

   }

   if ((charge==1)&&(int_bus_status == 3 ))    //Boost rectifer PWM charge
   {
     statcom_status=2;
     discharge=0;
     ac_disconnect=0;
     ioutDQO = park(&ioutABC,cf_cos_ph,cf_sin_ph);
     id_ref=-3.0;
          iq_ref=0.0;
     g_nFBCtr++; //FB frequency is 1/2 of PLL frequency, 10kHz
     if( g_nFBCtr > 1 )
      {
                          g_nFBCtr = 0;
               g_nVdcAVG++;
          PIT_ID.pfPosInput =&id_ref;
          PIT_ID.pfNegInput = &ioutDQO.D;
          UpdatePIAW(&PIT_ID);       //output of this PI is duty cycle Dd
                          dutyDQO.D =PIT_ID.fOutput;
   //Dd
                    //iq loop
                      PIT_IQ.pfPosInput = &iq_ref; //Pointer to iq Refference (Iq
command)
                          PIT_IQ.pfNegInput = &ioutDQO.Q;
                          UpdatePIAW(&PIT_IQ)//Output of this PI duty cycle Dq
          dutyDQO.Q =PIT_IQ.fOutput;
          dutyDQO.O =0.0;
          Inverter_out = Optimal_PWM (dutyDQO);
          // for 3-level, transfer the Switching state in the series protol format
          *SW_state_A = (Inverter_out.A.H1.L1 << 1) |(Inverter_out.A.H1.L2) |
          (Inverter_out.B.H1.L1 << 3)|(Inverter_out.B.H1.L2 <<2)|
```

```
(Inverter_out.C.H1.L1 << 5)|(Inverter_out.C.H1.L2 << 4);
            *SW_state_C = 0x0;
           LPF_id.Input=ioutDQO.D;
                    LPF(&LPF_id);
                    LPF_iq.Input=ioutDQO.Q;
                    LPF(&LPF_iq);
          if (g_nPWMCharge<2001)   // graph sample frequency is =20/2=5kHz
                    {
                    g_nPWMCharge++;
                VA[g_nPWMCharge%1000]=dutyDQO.D;
              VB[g_nPWMCharge%1000]=dutyDQO.Q;
              VC[g_nPWMCharge%1000]=E;
              Ia[g_nPWMCharge%1000]=dcbus_average;
              Ib[g_nPWMCharge%1000]=dutyDQO.D;
                    }
             if (g_nVdcAVG>9)
             {
               g_nVdcAVG=0;
                                DC_average.A = DC_average.A + (dcbus.A -
aver_A[dc_index] ) * 0.1f;
                                DC_average.B = DC_average.B + (dcbus.B -
aver_B[dc_index] ) * 0.1f;
                                DC_average.C = DC_average.C + (dcbus.C -
aver_C[dc_index] ) * 0.1f;
                                aver_A[dc_index] = dcbus.A;
                                aver_B[dc_index] = dcbus.B;
                                aver_C[dc_index] = dcbus.C;
                                dc_index = dc_index + 1;
                                if (dc_index >9) {
                                        dc_index = 0;
                                }
            if ((DC_average.A > Vdc_pwmc) && (DC_average.B > Vdc_pwmc) &&
(DC_average.C > Vdc_pwmc))
                    {
                     if(DCstat < 3)   //
                      {
                        DCstat++;
                      }
                    }
                    else
                    {
                     DCstat=0;
                    }
```

```
if(DCstat>2)

 {
                int_bus_status=4;
                g_nService = g_nPWMCharge;
              }
            }
     }

 }

 if ((int_bus_status == 4)&&(charge == 1))// close voltage loop
  {

  g_nFBCtr++;
   ioutDQO = park(&ioutABC,cf_cos_ph,cf_sin_ph);
  ret=0;
     if( g_nFBCtr > 1 )
                    {
                      LPF_id.Input=ioutDQO.D;
                      LPF(&LPF_id);
                      LPF_iq.Input=ioutDQO.Q;
                      LPF(&LPF_iq);
                            g_nFBCtr = 0;
                   DCcounta++;
                   DCcountb++;
                   DCcountc++;
                   DC_A= dcbus.A;
                   DC_B= dcbus.B;
                   DC_C= dcbus.C;
                   if (DCcounta>9)      // individual DC loop A  1000 Hz
                      {
                                    DCcounta=0;
                                    PI_DC_A.pfPosInput = &vd_ref;

                     PI_DC_A.pfNegInput = &DC_A;
                     UpdatePIAW(&PI_DC_A);
                     if (iq_ref >=-4.34)
                     {
                       delta_A = -1.0f*PI_DC_A.fOutput;
                     }
                     else {delta_A = PI_DC_A.fOutput;}
                     }
```

```
if (DCcountb>9)      // individual DC loop B
                    {
                                DCcountb=0;
                                PI_DC_B.pfPosInput = &vd_ref;

                    PI_DC_B.pfNegInput = &DC_B;
                    UpdatePIAW(&PI_DC_B);
                    if (iq_ref >=-4.34)
                    {
                      delta_B = -1.0f*PI_DC_B.fOutput;
                    }
                    else {delta_B = PI_DC_B.fOutput;}
                    }
                            if (DCcountc>9)      // individual DC loop C
                    {
                                DCcountc=0;
                                PI_DC_C.pfPosInput = &vd_ref;

                    PI_DC_C.pfNegInput = &DC_C;
                    UpdatePIAW(&PI_DC_C);
                    if (iq_ref >=-4.34)
                    {
                      delta_C = -1.0f*PI_DC_C.fOutput;
                    }
                    else {delta_C = PI_DC_C.fOutput;}
                    }
                            PIT_VD.pfPosInput = &dcbus_average;
                            PIT_VD.pfNegInput =&vd_ref;
                            UpdatePIAW(&PIT_VD);

                            //-----------------Id loop-------------------------



                            PIT_ID.pfPosInput = &PIT_VD.fOutput;
                  PIT_ID.pfNegInput = &ioutDQO.D;
                  UpdatePIAW(&PIT_ID);

                            dutyDQO.D = PIT_ID.fOutput;
                   iq_ref=Iqref*0.01f;
                  if (iq_ref<-8.0)
                    {
                      iq_ref=-8.0;
                    }
```

```
                if (iq_ref>8.0)
                  {
                   iq_ref=8.0;
                  }
                                PIT_IQ.pfPosInput = &iq_ref;

                PIT_IQ.pfNegInput = &ioutDQO.Q;
                                UpdatePIAW(&PIT_IQ);
                dutyDQO.Q =PIT_IQ.fOutput;
                dutyDQO.O =0.0;
                Inverter_out = Optimal_PWM (dutyDQO);
                *SW_state_A = (Inverter_out.A.H1.L1 << 1) |(Inverter_out.A.H1.L2) |
                (Inverter_out.B.H1.L1                <<                3)|(Inverter_out.B.H1.L2
<<2)|(Inverter_out.C.H1.L1 << 5)|
                (Inverter_out.C.H1.L2 << 4);
                *SW_state_C = 0x0;
                statcom_status=15;
                                  ttt++;
                   if ((ttt > 1)&&(g_nService < 20001)&&(Iqref>=-4.0))
                   {
                   ttt=0;
                   g_nService++;
                   VA[g_nService%1000]=ioutABC.A;
                   VB[g_nService%1000]=ioutABC.B;
                   VC[g_nService%1000]=ioutABC.C;
                   Ia[g_nService%1000]=delta_B;
                   Ib[g_nService%1000]=delta_C;
                    DCloopcount++;
                   }
                 else if (g_nService>10000)
                {
                  g_nService=0;
                }
                   if (discharge==1)
                   {
                   int_bus_status=5;
                   g_nDischarge=g_nService;
                   }
                          }
        }
        //Mode #4: AC discharge
        if ((discharge==1) && (int_bus_status==5) ){

           statcom_status=4;
```

```
    charge=0;
        ac_disconnect=0;
         run=0;
        ioutDQO = park(&ioutABC,cf_cos_ph,cf_sin_ph);
         g_nFBCtr++;
         if( g_nFBCtr > 1 )
          {
                        g_nFBCtr = 0;
             id_ref=1.0;
                    iq_ref=-0.001;
                    PIT_ID.pfPosInput = &id_ref;
             PIT_ID.pfNegInput = &ioutDQO.D;
             UpdatePIAW(&PIT_ID);
                        dutyDQO.D = PIT_ID.fOutput;
                        PIT_IQ.pfPosInput = &iq_ref;
                    PIT_IQ.pfNegInput = &ioutDQO.Q;
                        UpdatePIAW(&PIT_IQ);

                        dutyDQO.Q =PIT_IQ.fOutput;
                    dutyDQO.O =0.0;
                    Inverter_out = Optimal_PWM (dutyDQO);
            // for 3-level, transfer the Switching state in the series protol format
                    *SW_state_A = (Inverter_out.A.H1.L1 << 1) |(Inverter_out.A.H1.L2)
                     |      (Inverter_out.B.H1.L1      <<      3)|(Inverter_out.B.H1.L2
<<2)|(Inverter_out.C.H1.L1 << 5)|
                     (Inverter_out.C.H1.L2 << 4);
                    *SW_state_C = 0x0;
             if( DCstat >3)
               {
                 int_bus_status=3;
                  statcom_status=16;
                 ret=0;
                  discharge=0;
                  *SW_state_A = 0x3F;
                  *SW_state_C = 0x0;
                  return ret;
               }
            if (g_nVdcAVG>9)
             {
              g_nVdcAVG=0;
                            DC_average.A    =    DC_average.A    +    (dcbus.A    -
aver_A[dc_index] ) * 0.1f;
                            DC_average.B    =    DC_average.B    +    (dcbus.B    -
aver_B[dc_index] ) * 0.1f;
```

```
                    DC_average.C = DC_average.C + (dcbus.C -
aver_C[dc_index] ) * 0.1f;
                    aver_A[dc_index] = dcbus.A;
                    aver_B[dc_index] = dcbus.B;
                    aver_C[dc_index] = dcbus.C;
                    dc_index = dc_index + 1;
                    if (dc_index >9) {
                              dc_index = 0;
                    }
          if ((DC_average.A < Vdc_discharge) && (DC_average.B < Vdc_discharge)
          && (DC_average.C < Vdc_discharge))
               {
                 if(DCstat < 4)   //
                  {
                    DCstat++;
                  }
               }
               else
               {
                 DCstat=0;
               }
          }
     }
   }

   //Mode #6: AC disconnect
   if (ac_disconnect==1){
     ret=0;
     charge=0;
     statcom_status=5;
     int_bus_status=1; // back to the initial state
     }
   //Mode #5: stop
   if ((stop==1)){
      *SW_state_A = 0x3F;
      *SW_state_C = 0x0;
      int_bus_status = 1;
     ret=0;
      ac_disconnect=0;
      charge=0;
       statcom_status=6;
      *pDIGIOin = 0x006;
       return ret;
                    }
```

```
return(ret);                    //Program never comes to this line.
} /* end appl_process */

// ****** OPWM function,creating switching state of CMC by M and phase *****

static SW_forInverter Optimal_PWM (const TVA_DQO_DATA DQO)
{
    SW_forInverter Inverte_ret;
    float angle_shift_A;
    float angle_shift_B;
    float angle_shift_C;
    float temp1, temp2, temp3, temp4;
    temp1 = fabs(ftheta_ph);
    temp2 = fabs(ftheta_ph-g_cfpi12);
    temp3 = fabs(ftheta_ph-g_cfPi);
    temp4 = fabs(ftheta_ph-g_cfpi32);
    M = (float)sqrt(DQO.D*DQO.D+DQO.Q*DQO.Q);
    angle_shift_A = (float) atan2(DQO.Q,DQO.D)+ delta_A;
    angle_shift_B = angle_shift_A - g_cfPi23+ delta_B;
    angle_shift_C = angle_shift_A + g_cfPi23+ delta_C;
    phase_lock_loop(&vpccABC);
    Inverte_ret.A=Phase_PWM (M, angle_shift_A);
    Inverte_ret.B=Phase_PWM (M, angle_shift_B);
    Inverte_ret.C=Phase_PWM (M, angle_shift_C);
   return Inverte_ret;
}

static SW_forPhase Phase_PWM (const float M, const float angle_shift)
// M is modulation index = Amplitude_Vout_phase / Vdc, M = [0,1.15]
{
    SW_forPhase Phase_ret;
    float moving_angle;
    float M_rev;
    ang_Hbridge stable_angle;
    short index;
// when the moving_angle is greater than pi, then polar is -1, else it's 1
    short polar_phase =1;
    short compare_ret;
    short phase_output;
    short moving_angle_rev;
//PLL based cos; this funtion based on sin
    moving_angle = angle_shift + (float)ftheta_ph -  g_cfpi12;
// normalize the moving_angle to [0, 2*pi]
```

```
if (moving_angle < - g_cf2Pi)
    {
        moving_angle = moving_angle + g_cf2Pi;
    }

    if (moving_angle >= 2 * g_cf2Pi)
    {
         moving_angle = moving_angle - g_cf2Pi;
    }
    if (moving_angle <0)
    {
        moving_angle = moving_angle + g_cf2Pi;
    }
    if (moving_angle >= g_cf2Pi)
    {
         moving_angle = moving_angle - g_cf2Pi;
    }
// normalize the moving_angle to [0, pi] and specify polar value
    if (moving_angle >= g_cfPi)
        {
            moving_angle = moving_angle - g_cfPi;
            polar_phase = -1;
        }
// normalize the moving_angle to [0, pi/2]

    if (moving_angle >= g_cfpi12)
        {
            moving_angle = g_cfPi - moving_angle ;
        }
// convert moving angle to from rad to degree.
// To match the 3 level angle table, also need to times 100
    moving_angle_rev = (short) (moving_angle*5729.57805f);
// scale down M to [0, 0.91]
    M_rev = M*7.853981634e-1f;//M*g_cfPi*0.25f;
    M_test = M_rev;
    if(M_rev < 0.01) {M_rev = 0.01;}
    if(M_rev > 0.87) {M_rev = 0.87;}
    index = (short) (M_rev*100);
    stable_angle.ag0 = three_level_angle[index][0];
    stable_angle.ag1 = three_level_angle[index][1];
    stable_angle.ag2 = three_level_angle[index][2];
    stable_angle.ag3 = three_level_angle[index][3];
    stable_angle.ag4 = three_level_angle[index][4];
```

```
        if (( moving_angle_rev>= 0.0 )&&( moving_angle_rev < stable_angle.ag0))
                {
                        compare_ret = 0;
                }
        if (( moving_angle_rev>= stable_angle.ag0 )&&( moving_angle_rev <
stable_angle.ag1))
                {
                        compare_ret = 1;
                }
        if (( moving_angle_rev>= stable_angle.ag1 )&&( moving_angle_rev <
stable_angle.ag2))
                {
                        compare_ret = 0;
                }
        if (( moving_angle_rev>= stable_angle.ag2 )&&( moving_angle_rev <
stable_angle.ag3))
                {
                        compare_ret = 1;
                }
        if (( moving_angle_rev>= stable_angle.ag3 )&&( moving_angle_rev <
stable_angle.ag4))
                {
                        compare_ret = 0;
                }
        if ( moving_angle_rev>= stable_angle.ag4 )
                {
                        compare_ret = 1;
                }
       phase_output = compare_ret * polar_phase;
       if ( phase_output== 1)
       {
               Phase_ret.H1.L1=1;
               Phase_ret.H1.L2=0;
       }
       if ( phase_output== 0)
       {
               Phase_ret.H1.L1=0;
               Phase_ret.H1.L2=0;
       }
       if ( phase_output== -1)
       {
               Phase_ret.H1.L1=0;
               Phase_ret.H1.L2=1;
       }
```

```
 return Phase_ret;
}
```

4. STATCOM Model “statcom_model.c”

```
// STATCOM on-line model
#include "tva_types.h"
void statcom(const TVA_DQO_DATA duty, const float delta_t, STATCOM_PARAM*
prSTATCOM){
const float c=2700e-6f; // DC capacitance
const float L=2.5e-3f; // Coulping inductance
const float R=31.5e-3f;  // Coulping resistance(ESR of inductor)
const float vsd=49.79960944f;  // D-channel grid voltage (1.22*phase_va_pk)
const float omega=376.9911f;// 2*pi*60
float L_reverse = (float) (1/L);
float delta_v;       // voltage difference
float delta_id; // D-channel current
float delta_iq; // Q-channel current
float delta_vtemp1 = duty.D*(prSTATCOM->id);
float delta_vtemp2 = duty.Q*(prSTATCOM->iq);
// capacitor voltage
delta_v=(delta_vtemp1+delta_vtemp2)*delta_t/(-3*c);
// D-channel current
delta_id=((duty.D*(prSTATCOM->e)-vsd)*L_reverse+omega*(prSTATCOM->iq)-
(R*L_reverse)*(prSTATCOM->id))*delta_t;
delta_iq=(duty.Q*(prSTATCOM->e)*L_reverse-omega*(prSTATCOM->id)-
(R*L_reverse)*(prSTATCOM->iq))*delta_t;
prSTATCOM->e=(prSTATCOM->e)+delta_v;
prSTATCOM->id=(prSTATCOM->id)+delta_id;
prSTATCOM->iq=(prSTATCOM->iq)+delta_iq;
}
```

5. Digital Phase Lock Loop. “TVA_PLL.c”.

```
    // PLL function
#include <math.h>
#include "tva_control.h"
#include "tva_consts.h"
#include "tva_types.h"
extern float ftheta;                    //Theta of line voltages phase "AB"
extern float cf_cos;                    //Cosin from Sine table
extern float cf_sin;                    //Sin from Sine table
extern float ftheta_ph;                 //Theta of line voltages phase "A"
```

```
extern float cf_cos_ph;                    //Cosin from Sine table
extern float cf_sin_ph;                    //Sin from Sine table
extern PI_PARAMS PI_Vpccq;                 //Struture of PI compensator of PLL loop
extern PI_PARAMS PI_Omega;                 //Struture of PI compensator of PLL loop
static float omega_ref=-376.991;          //reference omega = 2*pi*60Hz
static float vpccq;
static float vpccq_ref=0;
void phase_lock_loop(TVA_ABC_DATA* abc_param)
    {
    TVA_DQO_DATA vpccDQO; //Voltage at the point of common coupling in DQO
    TVA_ABC_DATA theta1;  // theta1.A = ftheta; theta1.B = cos(ftheta); theta1.C =
sin(ftheta);
    TVA_ABC_DATA theta2;  // theta2.A = ftheta_ph;
    float pll_alpha;
    float pll_beta;
    pll_alpha = (float) ( abc_param->A-0.5f*(abc_param->B + abc_param->C) );
    pll_beta = (float) ( g_cfSqt32*(abc_param->B - abc_param->C) );
    vpccDQO.Q = -pll_alpha* ((float) cf_sin) + pll_beta* ((float) cf_cos);
    vpccq = vpccDQO.Q;          //Store VpccQ
    //Vq loop
    PI_Vpccq.pfPosInput = &vpccq;              //Pointer to Vpccq Refference (=0)
    PI_Vpccq.pfNegInput = &vpccq_ref;//Pointer      to      Vpccq      from      ABC->Q
transformation
    UpdatePI(&PI_Vpccq);                       //Output      of      this      PI      fn      is
PI_Vpccq.foutput (Omega)
    //Omega loop
    PI_Omega.pfPosInput = &PI_Vpccq.fOutput; //Pointer to output of the previous PI
(Omega)
    PI_Omega.pfNegInput = &omega_ref;          //Pointer      to      Omega      refference
(is 377 for 60Hz)
    UpdatePI(&PI_Omega);                        //Output      of      this      PI      fn      is
PI_Omega.foutput (Theta)
    if (PI_Omega.fOutput >= g_cf2Pi)           //Reset Theta
    {
        PI_Omega.fStatePrev = 0;
    }
    ftheta = PI_Omega.fOutput;
    theta1.A = ftheta;
    cos_sin(&theta1);
    cf_cos = theta1.B;
    cf_sin = theta1.C;
    //Phase angle calculation
    ftheta_ph = ftheta-g_cfPi16;  //30 degree lag phase shifting
    if (ftheta_ph <= 0)                        //Reset Theta if it let than 2*Pi
```

```
{
        ftheta_ph = g_cf2Pi+ftheta_ph;
    }
    theta2.A = ftheta_ph;
    cos_sin(&theta2);
    cf_cos_ph = theta2.B;
    cf_sin_ph = theta2.C;

}
```

6. Control Functions, “TVA_CONTROL.c”.

```
    #include <math.h>
#include "tva_types.h"
#include "tva_consts.h"
// Park's Transformation
TVA_DQO_DATA park(const TVA_ABC_DATA* pcrData,const float cf_cos, const float
cf_sin)
{
    TVA_DQO_DATA ret;
    float alpha_compt;
    float beta_compt;
    float gamma_compt;
    alpha_compt = (float) ( g_cfSqt23*(pcrData->A - 0.5*(pcrData->B + pcrData-
>C)) );
    beta_compt = (float) ( g_cfSqt23*g_cfSqt32*(pcrData->B - pcrData->C) );
    gamma_compt = (float) ( g_cfSqt13*(pcrData->A + pcrData->B + pcrData->C) );

    ret.D = alpha_compt*cf_cos+beta_compt*cf_sin;
    ret.Q = beta_compt*cf_cos-alpha_compt*cf_sin;
    ret.O = gamma_compt;
    return ret;
}

// Invert Park's Transformation
TVA_ABC_DATA inv_park(const TVA_DQO_DATA* pcrData,const float cf_cos,const
float cf_sin)
{
    TVA_ABC_DATA ret;
    float alpha_compt;
    float beta_compt;
    float gamma_compt;
    alpha_compt = (float) ( pcrData->D*cf_cos-pcrData->Q*cf_sin );
```

```
    beta_compt = (float) ( pcrData->D*cf_sin+pcrData->Q*cf_cos );
        gamma_compt = (float) ( pcrData->O );
        ret.A = (float) ( g_cfSqt23*(alpha_compt + g_cfSqt12*gamma_compt) );
        ret.B = (float) ( g_cfSqt23*(-0.5f*alpha_compt
        +g_cfSqt32*beta_compt+g_cfSqt12*gamma_compt) );
        ret.C = (float) ( g_cfSqt23*(-0.5f*alpha_compt-g_cfSqt32*beta_compt
        +g_cfSqt12*gamma_compt) );
        return ret;
    }

    //PI compensator
    void UpdatePI( PI_PARAMS* prPI )
    {
        float fErr = (float) ( (*(prPI->pfPosInput)-*(prPI->pfNegInput)) );
        prPI->fStatePrev = (float) ( (prPI->fStatePrev) + (prPI->fKe_1)*(prPI->fErrPrev)
+
        (prPI->fKe*fErr) );
        prPI->fErrPrev = (float) ( fErr );
        prPI->fOutput = (float) ( prPI->fStatePrev );

    }

    //PI-antiwindup compensator
    void UpdatePIAW( PIAW_PARAMS* prPI)
    {
        float ak = 0;
        float fErr = (float) ( (*(prPI->pfPosInput)-*(prPI->pfNegInput)) );
        prPI->fStatePrev = (float) ( (prPI->fStatePrev) + (prPI->fKp)*(fErr-(prPI-
>fErrPrev)) +
        ((prPI->fKii)*(fErr+(prPI->fErrPrev))) );
        if ((prPI->fStatePrev)>((prPI->uplimit)))
        {
                ak = ((prPI->fStatePrev)-(prPI->uplimit))*(prPI->fKanti);
                prPI->fOutput = prPI->uplimit;
        }
        else if ((prPI->fStatePrev)<((prPI->lolimit)))
        {
                ak = ((prPI->fStatePrev)-(prPI->lolimit))*(prPI->fKanti);
                prPI->fOutput = prPI->lolimit;
        }
        else
        {
                ak = 0;
                prPI->fOutput = prPI->fStatePrev;
```

```
}

    prPI->fStatePrev = (prPI->fStatePrev) - ak;
    prPI->fErrPrev = fErr;
}

void LPF( LPF_PARAMS* RMSInput)
{
   RMSInput->Output=(RMSInput->t1)*(RMSInput->Output)+
   (RMSInput->t2)*(RMSInput->Input)+(RMSInput->t3)*(RMSInput->preInput);
   RMSInput->preInput=(RMSInput->Input);
}
```