CHAPTER 1

INTRODUCTION

1.1 Background

A distributed controllers system (DCS) refers to a control system usually of a process plant /manufacturing system in which the different functions to the controllers are assigned according to the process plant requirements. Here, the controller elements are not central in location (like the brain) but are distributed throughout the system with each component sub-system controlled by one or more controllers. The entire system of controllers is connected by networks for communication and monitoring. DCS is used in the industries, to monitor and control distributed position equipment and process plant. A DCS typically uses custom designed processors as controllers and uses both proprietary interconnections and communications protocol for communication. Input and output modules form component parts of the DCS. The processor receives information from input modules and sends information to output modules. The input modules receive information from input instruments in the process (or field) and transmit instructions to the output instruments in the field. Computer buses or electrical buses connect the processor and modules through multiplexer or demultiplexers. Buses also connect the distributed controllers with the central controller and finally to the Human–machine interface (HMI) or control consoles.

The elements of a DCS may connect directly to physical equipment such as switches, pumps and valves or they may work through an intermediate system such as a SCADA system.

Distributed control systems (DCSs) are dedicated systems used to control manufacturing processes that are continuous or batch-oriented, such as oil refining, petrochemicals, central station power generation, fertilizers, pharmaceuticals, food and beverage manufacturing, cement production, steelmaking, and papermaking. DCSs are connected to sensors and actuators and use set point control to control the flow of material through the plant. The most common example is a set point control loop consisting of a pressure sensor, controller, and control valve. Pressure or flow measurements are transmitted to the controller, usually through the aid of a signal conditioning input/output (I/O) device. When the measured variable reaches a certain point, the controller instructs a valve or actuation device to open or close until the fluidic flow process reaches the desired set point.

Large oil refineries have many thousands of I/O points and employ very large DCSs. Processes are not limited to fluidic flow through pipes, however, and can also include things like paper machines and their associated quality controls , variable speed drives and motor control centres, cement kilns, mining operations, ore processing facilities, and many others.

Some of the properties of DCS are:

- Electrical power grids and electrical generation plants
- Environmental control systems
- Traffic signals
- Radio signals
- Water management systems
- Oil refining plants
- Metallurgical process plants
- Chemical plants

- Pharmaceutical manufacturing
- Sensor networks
- Dry cargo and bulk oil carrier ships

Existing distributed controllers system

1.1.1 Large-scale DCS

Distributed control systems are computer-based control systems in which the main components are located in different places. These components interact with each other via a local area network (LAN).

There are off-the-shelf DCSs designed and manufactured by Honeywell [Honeywell], Siemens [Siemens], and Emerson Process Management [Emerson], etc. Each vendor has its own method and communication protocol for providing the information and control.

Vendors also provide the required control for modern industrial systems and some can provide communication links to network systems such as Control net, Ethernet, Device net, Profi bus, ASI bus, Foundation Fieldbus, Data Highway/Remote I/O, Hart Protocol, and Modbus RTU . These DCSs are designed for process controls in petrochemical, pulp, food, beverage, and mining industries.

A typical example is Delta V Digital Automation System [Emerson] as illustrated in Figure 1.1. The field devices, such as sensors, valves, motors, and pumps, are connected with different digital communication buses like Fieldbus, HART, and DeviceNet. The linking devices are gateways or bridges between the field buses and high speed Ethernet.

Operator stations are Windows workstations and server-based PCs. Overall, the DeltaV digital automation system is a DCS based on Windows workstations and server-based PCs, Ethernet technology, and bus standards.

The DeltaV suite of engineering tools handles configuration management both locally and remotely for all aspects of the DeltaV system and intelligent field devices. A global and centralized configuration database coordinates control strategies, process graphics, history, events, and change management. All DeltaV hardware is automatically recognized as it is plugged in. The DeltaV operation software provides an easy-to-use environment for process operations and information access.

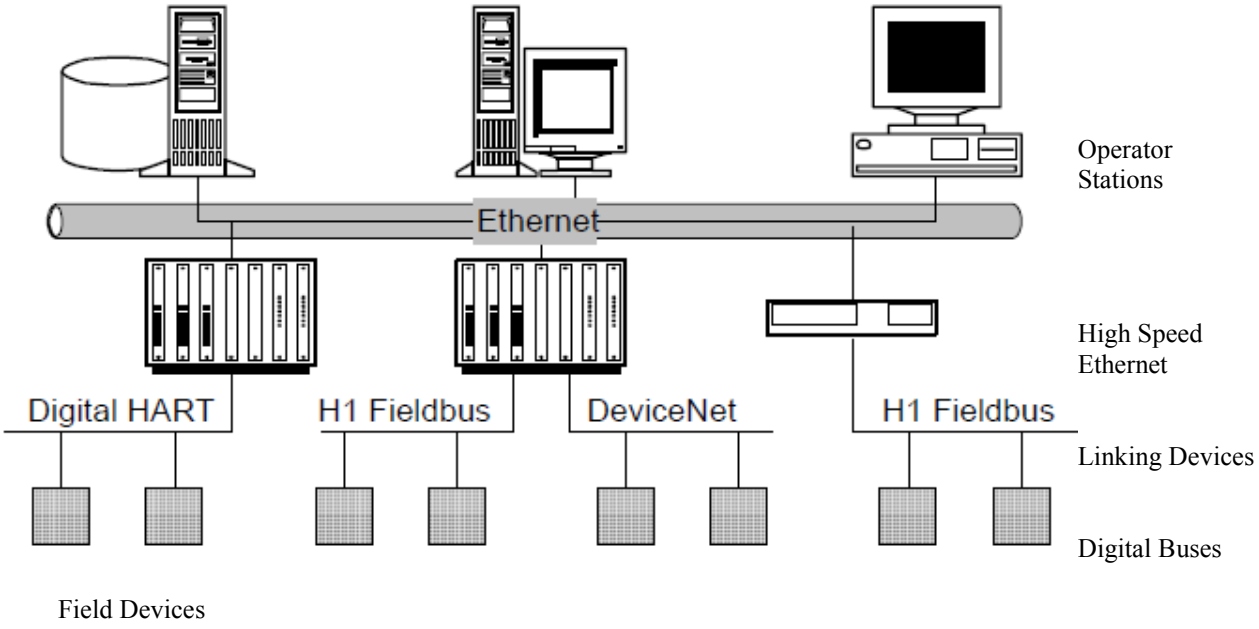

Figure 1.1 DeltaV architecture

All operations applications are fully remotely accessible anywhere on an Ethernet network or via modem. The process history view provides continuous trend, event views and batch views to intuitively present these different types of historical information. By using digital plant architecture (PantWeb) and plant-wide asset optimization software (AMS), the DeltaV provides easy access to

vital device information for calibration configuration, devices audit trail, and advanced diagnostics for predictive maintenance.

Large-scale DCSs, like DeltaV, are usually used to control almost all processes in a big plant.

1.1.2 Small-scale DCSs

A small smart distributed control system is shown in Figure 1.2. The system includes an industrial PC, several controllers, and a communication interface using RS485. The PC is an operating station that is used to configure function blocks and monitor the process. The controller is used to sample data and execute control algorithms. Communication between the PC and the controllers is implemented in a master-slave type broadcasting on the communication network. The slave nodes are not allowed to transmit data without a request from the master, and do not directly communicate with each other. When a slave needs to send a message to another slave, the message has to be sent to the master and then the master forwards the message to the receiver.

Marti et al. proposed an integrated approach to real-time distributed control systems over Field bus . A control loop is implemented in a distributed architecture, with three nodes: a sensor node, a controller node and an actuator node that communicate with each other across a fieldbus communication network. The sensor node periodically samples the process and sends the data to the controller node.

The controller node executes a control algorithm and sends the output to the actuator node.

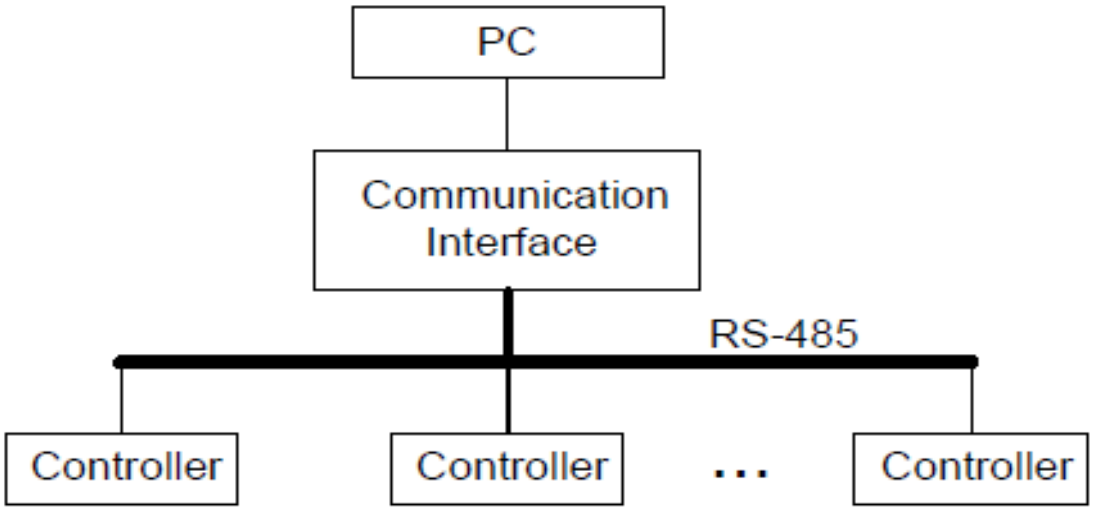

Figure 1.2 Small scale DCS

A distributed control system is introduced in for supervising a power supply system of a telecommunication system. The DCS consists of a main controller, several distributed units, and a communication network using multi-drop RS-485 bus with a proprietary communication protocol. The core of the supervision system is the main controller, which is a micro-controller based unit with communication ports (RS-232, RS-485) and a user interface. The RS-232 port is intended for higher-level supervision, typically for remote connection via modem or computer network. The RS-485 ports are used for communication between the main controller and distributed units. Each distributed unit contains a small micro-controller for data acquisition and control. Each unit has a unique address. They are connected to the bus and receive both serial communication and power from the bus. The main controller collects information from distributed units using the master-slave network and sends the messages to all distributed units, but only the right unit with the correct address will answer.

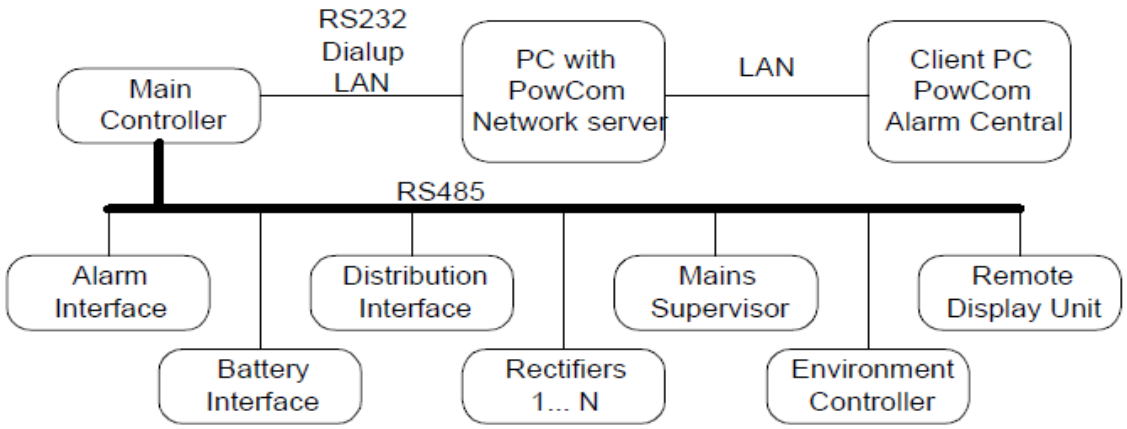

Figure 1.3 System architecture

1.1.3 Internet-based DCS

DCOM, CORBA, and Java RMI are well known middleware architectures based on object-oriented methodologies and have been extensively used in commercial and military distributed information systems. Since they support communication protocols that are transparent to operating systems, users can design and manage the whole system without paying too much attention to the data transmission between remote devices . CORBA-, DCOM-, and Tspace-based distributed control systems are designed and implemented for performance analysis using I/O object, control object, broadcasting service and event service. When using middleware as a communication bus for objects, the delay may be greater than the one introduced by the Fieldbus, since DCOM, CORBA, and Java RMI have not been designed for the physical device level but for integrating and linking of software components at higher levels.

1.2 Objective of Project

The objective of this project is to design a distributed controllers system over Ethernet bus for motion control drives using 8051 micro-controller and AVR micro-controller, respectively. The PWM technology is employed for generating the pulses and direct vector control method is used for induction motor drive where as simple PWM method is employed for Dc motor. Operator console is connected to Ethernet bus for system operator operations.

1.3 Scope of Project

In order to achieve the objective of the project, there are several scope had been outlined. The scope of this project includes using Distributed controllers system, to control DC drive and induction drive by PWM pulse-Width generated by using 8051 micro-controller and AVR micro-controller, to build hardware for the system.

1.4 Outline of Thesis

This thesis consists of VI chapters. In first chapter, it discusses about the objective and scope of this project. While Chapter II describes the literature reviews on controllers including programmable logic controller. It also discusses about features of the controllers. Chapter III presents a brief on communication protocol, discussion of different buses and comparison of different buses. Chapter IV discusses about drives and their algorithm that is implemented in this project. Chapter V discusses about the implementation and testing. And in chapter VI conclusion and further scope.

1.5 Conclusion

This chapter gives a brief introduction about Distributed controllers system (DCS) and need for DCSs. A brief status about existing Distributed control system has presented to show the need of DCS. Also objective of this thesis is brought out and brief outline of the project is presented.

CHAPTER 2

THEORY AND LITERATURE REVIEW

2.1 Introduction

In this chapter a brief literature review of various controllers and associated areas have been discussed without exhaustive elaboration. The review includes the Control system, Automatic controller system, distributed control system, Programmable logic controller, 8051 micro-controller and AVR micro-controller.

2.2 Control system

A control system is a device, or set of devices to manage, command, direct or regulate the behaviour of other devices or system. The control system is that means by which any quantity of interest in a machine, mechanism or other equipment is maintained or altered in accordance with a desired manner. The first Automatic control system, the fly-ball governor, to control the speed of steam engines, was invented by James Watt in 1977. It was about hundred years later that Maxwell analysed the dynamics of the fly-ball governor. The importance of positioning heavy masses like ships and guns quickly and precisely was realized during the world war 1. In 1922, Minorsky worked on automatic controllers for steering ships and showed how stability could be determined from the differential equations describing the system. In 1932, Nyquist developed a relatively simple procedure for determining the stability of closed-loop system .In 1934, the servomechanism term for position control systems was developed. During the Decade of 1940's mathematical and analytical methods were developed. During the world war 2 automatic aeroplane pilots, gun positioning systems, radar tracking system and other

military equipment based on feedback control principal was developed. The industrial use of automatic control has tremendously increased since the world war 2. Modern industrial processes such as manufacture and treatment of chemicals and metals are now automatically controlled.

2.3 Distributed Controllers System

The DCS was introduced in 1975. Both Honeywell and Japanese electrical engineering firm Yokogawa introduced their own independently produced DCSs. US-based Bristol also introduced their UCS 3000 universal controller in 1975. In 1978 Metso (known as Valmet in 1978) introduced their own DCS system called Damatic (latest generation named Metso DNA). In 1980, Bailey introduced the NETWORK 90 system. Also in 1980, Fischer & Porter Company introduced DCI-4000 (DCI stands for Distributed Control Instrumentation). The DCS largely came about due to the increased availability of microcomputers and the proliferation of microprocessors in the world of process control. In the early 1970s Taylor Instrument Company, developed the 1010 system, Foxboro the FOX1 system and Bailey Controls the 1055 systems. All of these were DDC applications implemented within minicomputers and connected to proprietary Input/Output hardware. Midac Systems, of Sydney, Australia, developed an objected-oriented distributed direct digital control system in 1982. The central system ran 11 microprocessors sharing tasks and common memory and connected to a serial communication network of distributed controllers each running two Z80s. The system was installed at the University of Melbourne. Digital communication between distributed controllers, workstations and other computing elements (peer to peer access) was one of the primary advantages of the DCS. The traditional DCS suppliers introduced new generation DCS System based on the latest Communication and IEC Standards, which resulting in a trend of combining the traditional concepts/functionalities for PLC and DCS into a one for all solution—named "Process Automation System". The current

next evolution step is called Collaborative Process Automation Systems. Distributed control systems (DCSs) are used to control manufacturing processes such as oil refining, petrochemicals, central station power generation, fertilizers, pharmaceuticals, food and beverage manufacturing, cement production, steelmaking, and papermaking. DCSs are connected to sensors and actuators and use set point control to control the flow of material through the plant. Application of DCSs are also in paper machines , quality controls , variable speed drives and motor control centres, cement kilns, mining operations, ore processing facilities, and many others. Modern DCSs also support neural networks and fuzzy application. DCSs may employ one or more workstations and can be configured at the workstation or by an off-line personal computer. Local communication is handled by a control network with transmission over twisted pair, coaxial, or fibre optic cable. A server and/or applications processor may be included in the system for extra computational, data collection, and reporting capability.

Some of the advantages of DCS are as:

1. Overall cost of the installation is lower.
2. Interface with the process is improved.
3. More reliable.
4. Flexible and relatively easy to expand.
5. Programming required to tailor the system can be done without knowing a high-level programming language.

2.4 Programmable Logic Controllers

A programmable logic controller (PLC) or programmable controller is a digital computer used for automation of electromechanical processes, such as control of machinery on factory assembly lines, amusement rides, or light fixtures. PLCs are used in many industries and machines. Unlike general-purpose computers, the PLC is designed for multiple inputs and output arrangements, extended temperature ranges, immunity to electrical noise, and resistance to

vibration and impact. Programs to control machine operation are typically stored in battery-backed-up or non-volatile memory. A PLC is an example of a hard real time system since output results must be produced in response to input conditions within a limited time, otherwise unintended operation will result. The PLC was invented in response to the needs of the American automotive manufacturing industry. Programmable logic controllers were initially adopted by the automotive industry where software revision replaced the re-wiring of hard-wired control panels when production models changed. Before the PLC, control, sequencing, and safety interlock logic for manufacturing automobiles was accomplished using hundreds or thousands of relays, cam timers, drum sequencers, and dedicated closed-loop controllers. The process for updating such facilities for the yearly model change-over was very time consuming and expensive, as electricians needed to individually rewire each and every relay. Digital computers, being general-purpose programmable devices, were soon applied to control of industrial processes. Early computers required specialist programmers, and stringent operating environmental control for temperature, cleanliness, and power quality. Using a general-purpose computer for process control required protecting the computer from the plant floor conditions.

In 1968 GM Hydramatic (the automatic transmission division of General Motors) issued a request for proposal for an electronic replacement for hard-wired relay systems. The winning proposal came from Bedford Associates of Bedford, Massachusetts. The first PLC, designated the 084 because it was Bedford Associates' eighty-fourth project, was the result. Bedford Associates started a new company dedicated to developing, manufacturing, selling, and servicing this new product: Modicon, which stood for MOdular DIgital CONtroller. One of the people who worked on that project was Dick Morley, who is considered to be the "father" of the PLC. The Modicon brand was sold in 1977 to Gould Electronics, and later acquired by German Company AEG and

then by French Schneider Electric, the current owner. One of the very first 084 models built is now on display at Modicon's headquarters in North Andover, Massachusetts. It was presented to Modicon by GM, when the unit was retired after nearly twenty years of uninterrupted service. Modicon used the 84 moniker at the end of its product range until the 984 made its appearance.

2.5 Micro-controller

A microcontroller (sometimes abbreviated µC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

Some microcontrollers may use four-bit words and operate at clock rate frequencies as low as 4 kHz, for low power consumption (mill watts or microwatts). They will generally have the ability to retain functionality while waiting for an event such as a button press or other interrupt; power consumption while sleeping (CPU clock and most peripherals off) may be just

Nano watts, making many of them well suited for long lasting battery applications.

The first single-chip microprocessor was the 4-bit Intel 4004 released in 1971, with the Intel 8008 and other more capable microprocessors becoming available over the next several years. However, both processors required external chips to implement a working system, raising total system cost, and making it impossible to economically computerize appliances.

The engineers Gary Boone and Michael Cochran succeeded in creating the first microcontroller in 1971. The result of their work was the TMS 1000, which went commercial in 1974. It combined read-only memory, read/write memory, processor and clock on one chip and was targeted at embedded systems.
Partly in response to the existence of the single-chip TMS 1000, Intel developed a computer system on a chip optimized for control applications, the Intel 8048, with commercial parts first shipping in 1977. It combined RAM and ROM on the same chip. This chip would find its way into over one billion PC keyboards, and other numerous applications. At that time Intel's President, Luke J. Valenter, stated that the microcontroller was one of the most successful in the company's history, and expanded the division's budget over 25%.

Most microcontrollers at this time had two variants. One had an erasable EPROM program memory, which was significantly more expensive than the PROM variant which was only programmable once. Erasing the EPROM required exposure to ultraviolet light through a transparent quartz lid. One-time parts could be made in lower-cost opaque plastic packages.

In 1993, the introduction of EEPROM memory allowed microcontrollers (beginning with the Microchip PIC16x84) to be electrically erased quickly without an expensive package as required for EPROM, allowing both rapid prototyping, and In System Programming. The same year, Atmel introduced the first microcontroller using Flash memory. Other companies rapidly followed suit, with both memory types.

Cost has plummeted over time, with the cheapest 8-bit microcontrollers being available for under $0.25 in quantity (thousands) in 2009, and some 32-bit microcontrollers around $1 for similar quantities. Nowadays microcontrollers are cheap and readily available.

2.6 8051 Micro-controller The Intel 8051 microcontroller is one of the most popular general purpose microcontrollers in use today. The success of the Intel 8051 spawned a number of clones which are collectively referred to as the MCS-51 family of microcontrollers, which includes chips from vendors such as Atmel, Philips, Infineon, and Texas Instruments.The Intel 8051 is an 8-bit microcontroller which means that most available operations are limited to 8 bits. There are 3 basic "sizes" of the 8051: Short, Standard, and Extended. The Short and Standard chips are often available in DIP (dual in-line package) form, but the Extended 8051 models often have a different form factor, and are not "drop-in compatible". All these things are called 8051 because they can all be programmed using 8051 assembly language, and they all share certain features (although the different models all have their own special features).

2.6.1 Features

1. 4 KB on chip program memory.
2. 128 bytes on chip data memory(RAM).

3. 4 reg banks.
4. 128 user defined software flags.
5. 8-bit data bus
6. 6-bit address bus
7. 32 general purpose registers each of 8 bits
8. 16 bit timers (usually 2, but may have more, or less).
9. 3 internal and 2 external interrupts.
10.Bit as well as byte addressable RAM area of 16 bytes.
11.Four 8-bit ports, (short models have two 8-bit ports).
12.16-bit program counter and data pointer.
13.1 Microsecond instruction cycle with 12 MHz Crystal.

8051 models may also have a number of special, model-specific features, such as UARTs, ADC, OpAmps, etc... 8051 chips are used in a wide variety of control systems, telecom applications, robotics as well as in the automotive industry. By some estimations, 8051 family chips make up over 50% of the embedded chip market

2.7 AVR Microcontroller

The AVR is a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time.

The AVR architecture was conceived by two students at the Norwegian Institute of Technology (NTH) Alf-Egil Bogen and Vegard Wollan.The original AVR MCU was developed at a local ASIC house in Trondheim, Norway called Nordic VLSI at the time, now Nordic Semiconductor, where Bogen and Wollan

were working as students. It was known as a μRISC (Micro RISC) and was available as silicon IP/building block from Nordic VLSI. When the technology was sold to Atmel from Nordic VLSI, the internal architecture was further developed by Bogen and Wollan at Atmel Norway, a subsidiary of Atmel. The designers worked closely with compiler writers at IAR Systems to ensure that the instruction set provided for more efficient compilation of high-level languages. Atmel says that the name AVR is not an acronym and does not stand for anything in particular. The creators of the AVR give no definitive answer as to what the term "AVR" stands for. However, it is commonly accepted that AVR stands for Alf (Egil Bogen) and Vegard (Wollan)'s Risc processor. Among the first of the AVR line was the AT90S8515, which in a 40-pin DIP package has the same pin out as an 8051 microcontroller, including the external multiplexed address and data bus. The polarity of the RESET line was opposite (8051's having an active-high RESET, while the AVR has an active-low RESET) but other than that, the pin out was identical. The AVR is a modified Harvard architecture machine where program and data are stored in separate physical memory systems that appear in different address spaces, but having the ability to read data items from program memory using special instructions.

2.7.1 Features of AVR

1. Multifunction, bi-directional general-purpose I/O ports with configurable, built-in pull-up resistors
2. Multiple internal oscillators, including RC oscillator without external parts
3. Internal, self-programmable instruction flash memory up to 256 kB (384 kB on XMega)
4. In-system programmable using serial/parallel low-voltage proprietary interfaces or JTAG
5. Optional boot code section with independent lock bits for protection

6. Debug WIRE uses the /RESET pin as a bi-directional communication channel to access on-chip debug circuitry. It is present on devices with lower pin counts, as it only requires one pin.
7. Internal data EEPROM up to 4 kB
8. Internal SRAM up to 16 kB (32 kB on XMega)
9. 8-bit and 16-bit timers
10. PWM output (some devices have an enhanced PWM peripheral which includes a dead-time generator)
11. Input capture
12. Analog comparator
13. 10 or 12-bit A/D converters, with multiplex of up to 16 channels
14. 12-bit D/A converters
15. A variety of serial interfaces, including
16. I²C compatible Two-Wire Interface (TWI)
17. Serial Peripheral Interface Bus (SPI)
18. Universal Serial Interface (USI) for two or three-wire synchronous data transfer
19. Watchdog timer (WDT)
20. Multiple power-saving sleep modes

2.8 Conclusion

An extensive literature review of control system, distributed controllers system and Microcontrollers has been presented in this chapter.

Chapter 3

Communication Protocols and field buses

3.1 Introduction

This chapter describes the study of communication protocol , Field buses ,World FIP , Macro and communication protocol function and description. Therefore, this chapter will briefly review major industrial communication protocols and their most prominent features and compare them with the communication

3.2 Communication protocol

The communication protocol provides communication between the applications manager and hardware manager. It is designed to support any type of module and to provide real time data distribution.

Distributed controls have been extensively used in factory automation and motion control systems for the last twenty years. Increased modularity, fault tolerance, expandability and significantly improved overall system flexibility have constantly fuelled the drive for integration of factory elements into the computer integrated manufacturing environment. The idea is to employ hierarchical control over the whole manufacturing process, and integrate all factory elements into a coherent control architecture, as depicted in Fig. 3-1. All the protocols have been specifically tailored to meet targeted system requirements.

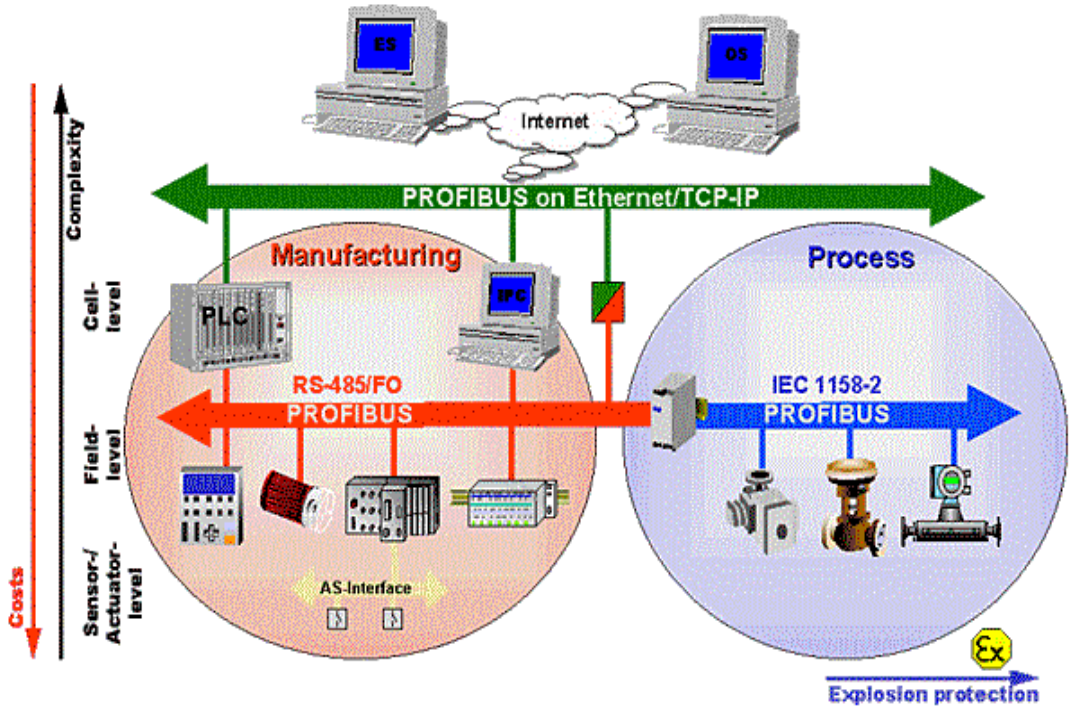

Fig. 3.1 Industrial communications for the example of the PROFIBUS standard

While industrial communication protocols provide distributed control and communications on a field level (connecting drives, PLCs, controllers, sensors, actuators, etc.). This would bring modular design, significant flexibility, hierarchical control and many other benefits on a converter control level, compared with industrial communications, in which hierarchical control starts above the converter and drive level.

In factory automation, drives and converters are treated as a building block rather than as whole new distributed system. This means that a distributed control has to operate on a much faster time scale, with a few orders of magnitude smaller time constants. Also, synchronization becomes much more of a pronounced issue due to the smaller time constants. Noise susceptibility and noise immunity of the distributed controller arises as another important issue. Since the communication link is getting physically close to the electro-

magnetic interference EMI sources (power converters), noise immunity of a control and communication system starts to play a key role in the system reliability and robustness.

3.3 Network topologies

The two most widely used network topologies in industrial communications are
(i) point-to-point serial interface links and
(ii) multi-point connection offered by the local area network (LAN).

At the present time, many motor drives are connected to their steering units using point-to-point serial interface links (e.g. RS 232 or RS 422). This approach suffers from two major drawbacks. First of all, one control unit must drive several drives and handle communication with them simultaneously, which can seriously overload the main controller and limit the response time.

This approach also limits the expandability of the control system. Due to low cost this approach feasible for many applications.

LAN offer multi-point connections and are much more flexible and application independent. This is the reason why most of today's control protocols are based on LANs. The basic requirement for LANs is that they have to be open and to support devices made by different manufacturers. This provides the system with high expandability and significant flexibility.
The LAN communication protocols for real-time control applications can be roughly divided into field buses and specific networks. The former is the general purpose network devised for field-level control that can connect drives, PLCs, PCs, I/O modules, sensors, and actuators. They are designed to provide

wider flexibility and to allow for interconnection of different units at the cost of speed and response time. Specific networks are less flexible and are designed to provide link between PLCs and controllers on one side, and sensors, drives and actuators on the other. Those specific networks are designed to provide faster response time and better synchronization at the cost of reduced flexibility.

3.4 Field Buses

Fieldbus is an industrial network system for real-time distributed control. It is a way to connect instruments in a manufacturing plant. Fieldbus works on a network structure which typically allows daisy-chain, star, ring, branch, and tree network topologies. Previously, computers were connected using RS-232 (serial connections) by which only two devices could communicate. This would be the equivalent of the currently used 4-20 mA communication scheme which requires that each device has its own communication point at the controller level, while the fieldbus is the equivalent of the current LAN-type connections, which require only one communication point at the controller level and allow multiple (hundreds) of analog and digital points to be connected at the same time. This reduces both the length of the cable required and the number of cables required. Furthermore, since devices that communicate through fieldbus require a microprocessor, multiple points are typically provided by the same device. Some fieldbus devices now support control schemes such as PID control on the device side instead of forcing the controller to do the processing.

There are disadvantages to using fieldbus compared to the 4-20 mA analog signal standard (4-20 mA with HART):

- Fieldbus systems are more complex, so users need to be more extensively trained or more highly qualified
- The price of fieldbus components is higher

- Fieldbus test devices are more complex compared to a (high-spec) multimeter that can be used to read and simulate analog 4-20 mA signals
- Slightly longer reaction times with fieldbus, depending on the system
- Device manufacturers have to offer different versions of their devices (e.g. sensors, actuators) due to the number of different (incompatible) fieldbus standards. This can add to the cost of the devices and to the difficulty of device selection and availability.
- One or more fieldbus standards may predominate in future and others may become obsolete. This increases the investment risk when implementing fieldbus.

3.4.1 PROFIBUS

Process Field BUS is the German proposal for the open field bus standard for a wide range of applications in manufacturing and process automation .
PROFIBUS is designed for both high-speed time critical applications and complex communication tasks. The protocol architecture is oriented to the reduced ISO OSI (open system interconnection) reference model in which each layer handles a precisely defined tasks.

Layer 1 (the physical layer) defines the physical transmission characteristics.
There are currently three transmission methods (physical profiles) available for PROFIBUS:
1. RS 485 transmission for universal applications in manufacturing automation
2. IEC 1158-2 transmission for use in process automation; and
3. optical fibers for improved interference immunity and large network distances

In the course of further technical developments, it is intended to use commercial Ethernet components with 10 Mb/s and 100 Mb/s as the physical layer for PROFIBUS.

Layer 2 (the data-link layer) defines the bus access protocol. PROFIBUS is designed to support two types of devices, namely master-devices and slave-devices as shown in Fig. 3-2. Master devices determine the data communication on the bus. Master can send messages without an external request whenever it holds the bus access rights (tokens). On the other side slave devices are peripherals such as I/O devices, valves, drives and measuring transducers. They can only acknowledge received data or send messages when asked to do so by the master. The medium access control protocol includes the token passing procedure among masters. Each master (upon receiving token) takes control over the bus and starts communicating with slaves and other masters.

After the token hold time expires the token is passed to the following master node which then assumes the control over the bus. This allows for the following system configurations:

1. pure master-slave system,
2. pure master-master system (token passing), and
3. Combination of the two.

The actual token hold time of a master depends on the configured token rotation time.

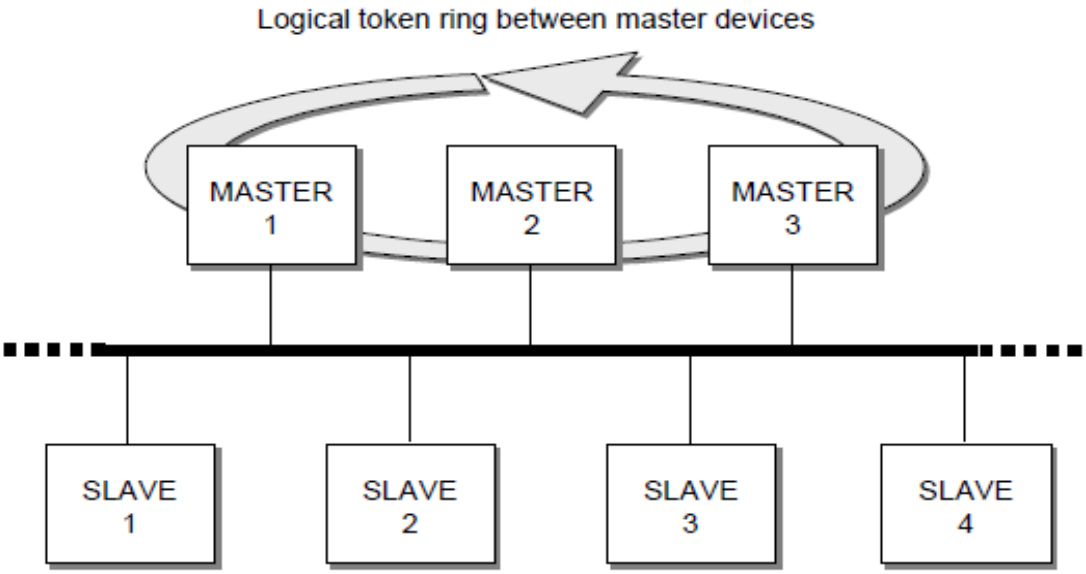

Fig. 3-2. PROFIBUS medium access control.

Layer 7 (the application layer) defines the application functions. It consists of two protocols, namely the DP protocol and FMS (field message service) protocol. DP is designed for efficient data exchange at the field level. This protocol is mostly used for cyclic data exchange between the automation system and distributed peripherals. FMS is the universal communication profile for demanding communication tasks. It allows object manipulation, with objects being variables, arrays, matrices, variable lists, program calls, subroutines and so on. It is divided into two sub-layers, i.e.. FMS (field bus message specification) and LLI (lower layer interface). The architecture of the PROFIBUS is shown in Fig. 3-2.

3.4.2 World FIP

Factory automation protocol is a French-Italian proposal for the field bus . It is designed to provide links between level zero (sensors/actuators) and level one (PLCs, controllers, etc.) in automation systems . World FIP is also designed

according to the reduced ISO/OSI reference model, meaning that all protocol functions are implemented within the physical data-link and
application layer. Although World FIP protocol supports both copper wire and optical fiber as a transmission media, twisted pair copper cable is the one used most often. The data link layer provides a medium access protocol. There is always one station, named the bus arbitrator (distributor; see Fig. 3-3), that handles the communication and bus access and a number of other stations which respond to the arbiter polling.

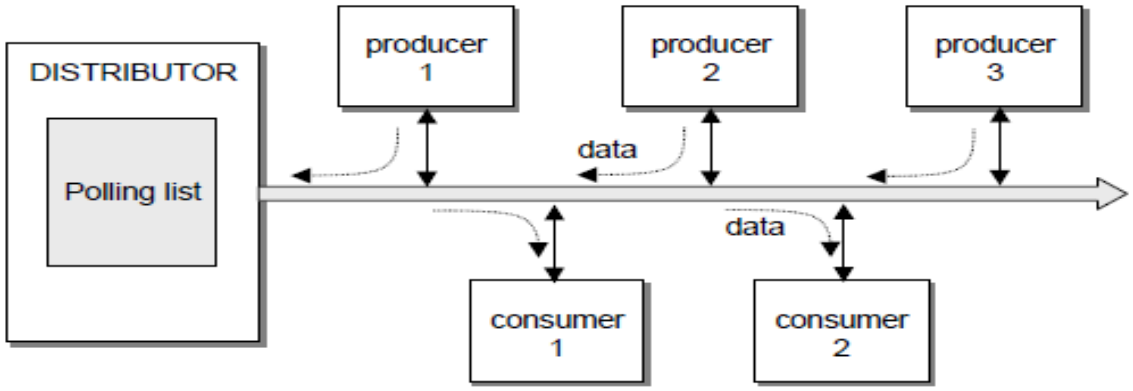

Fig. 3.3 Block diagram of World FIP medium access mechanism.

Each station on the bus has several produced and consumed buffers. When the arbitrator inserts the question command on the bus and states the buffer address the station that is producer of the buffer will respond by putting the buffer value on the bus. The stations that are consumers of that variable will respond with accepting the value from the bus and storing it. Data traffic consists of time-critical data (control variables) that are exchanged periodically and non-critical data such as messages. Cyclic variables always travel on time, regardless of other network traffic. Less time-dependent data (such as diagnostic reports), are sent via the message service. This is typically used for installing and setting applications, network supervision and diagnosis and integration with higher-

level systems. Unlike token-passing systems, World FIP leaves no doubt about transmission time and regularity for cyclic data variables. Word FIP applications layer is divided into two distinct groups:

• MPS (manufacturing periodical/a periodical services) and
• sub MMS (subset of messaging services).

The MPS application layer provides the user with local read/write services, remote read/write services, variable transmission/reception indications and information on the spatial and temporal consistency of the data. All those services are used for real time data distribution throughout the whole network. For example, the local read service provides the application layer with the variable from the consumed buffer that exists in the data link layer. The local read and write service uses the data-link layer services

Those services generate no traffic on the bus since those produced and consumed buffers reside in the data-link layer. MMS services are used for non-critical traffic such as system configuration, monitoring, etc.

3.4.3 CAN

The controller area network (CAN) is a field bus proposed by Bosch for automotive applications . It is a serial communications protocol, which efficiently supports distributed real-time control with a very high level of security. CAN follows the ISO/OSI reduced model, with bus topology, client-server model and an original medium access method. In all protocol messages there are no origin/destination addresses but only an identifier. Every station first reads the identifier and then decides whether or not to read the rest. The priority field of the frame indicates the type of message transmitted and its priority. When a station wants to transmit the message it has to compare its

relative priority to the network message priority. If it is less or equally important it has to wait until the bus is clear. This medium access control is a multi master protocol with distributed priority arbitration.

The advantage of CAN protocol is that it was developed for automotive applications thus being optimized from the cost point of view. Also, market availability of the supporting components is very good, which makes the whole standard widely accepted. The fact that this protocol was designed for the inexpensive copper wire transmission medium makes it less suitable for applications in which EMI noise is a significant issue.

3.5 Ethernet bus

Ethernet was developed in the late 1970's by the Xerox Corporation at their Palo Alto Research Centre in California. It has been estimated that over 70% of the worlds networks use the Ethernet protocol, so with this in mind it would seem only sensible to discuss how it works Ethernet uses a protocol called **CSMA/CD**, this stands for Carrier Sense, Multiple Access with Collision Detection **Carrier Sense** - When a device connected to an Ethernet network wants to send data it first checks to make sure it has a carrier on which to send its data (usually a piece of copper cable connected to a hub or another machine).

Multiple Access - This means that all machines on the network are free to use the network whenever they like so long as no one else is transmitting.

Collision Detection - A means of ensuring that when two machines sta transmit data simultaneously, that the resultant corrupted data is discarded, an transmissions are generated at differing time intervals.

3.6 Mod bus

Modbus is a serial communications protocol published by Modicon in 1979 for use with its programmable logic controllers (PLCs). Simple and robust, it has since become a de facto standard communication protocol, and it is now amongst the most commonly available means of connecting industrial electronic devices. The main reasons for the extensive use of Modbus in the industrial environment are:

- It has been developed with industrial applications in mind
- It is openly published and royalty-free
- It is easy to deploy and maintain
- It moves raw bits or words without placing many restrictions on vendors

Modbus allows for communication between many (approximately 240) devices connected to the same network, for example a system that measures temperature and humidity and communicates the results to a computer. Modbus is often used to connect a supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (SCADA) systems. Many of the data types are named from its use in driving relays: a single-bit physical output is called a *coil*, and a single-bit physical input is called a *discrete input* or a *contact*.

3.7. MICRO

MACRO is a communication standard for distributed machine control. MACRO stands for a motion and control ring optical and was designed for multiple-axis precision motion control . MACRO uses a ring topology to allow master controllers to communicate with both slave nodes and other master controllers. A MACRO network is organized as ring architecture with multiple masters and

slaves connected. Communication throughout the ring is started by a pr-designated ring-master.

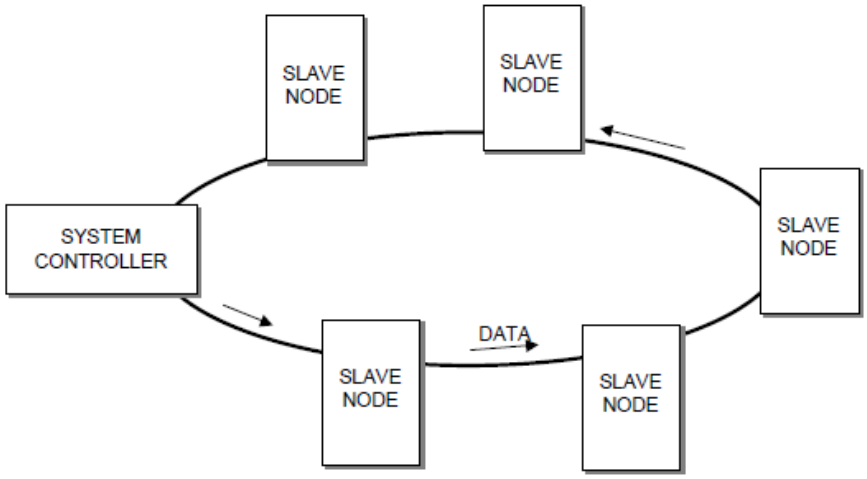

Fig. 3.4 MACRO ring network consists of a group of master and slave stations.

Communication is always originated by a master that sends a data packet (with appropriate address) down the stream. The node that receives the data packets checks the address and if it is the same as the local address, it takes the data from the packet and loads the data packet with local data and sends the packet down the ring. If the local address is different the node just passes the data to the next one in the ring. A master's ring hardware shifts all active node data from the master's transmission registers across the ring to matching input registers of the appropriate slave node. The slave's ring

hardware shifts the node data that is addressed to it into a set of receiving registers. Data is returned to the master at the same time the data is received from the ring by the slave's hardware transmission registers.

3.8 Communication Protocol Functional Description

The master-slave protocol insures deterministic response of the network. If an error occurs during transmission, corrupt data is not used. Instead, the new data

simply overwrite the previous data. This way the data flow is kept strictly predetermined. There are two basic types of information communicated through the control network:

i. Time-critical data (exchanged in every switching cycle) and

ii. Time non-critical data (transmitted only after all the critical time variables have been passed to all nodes).

Time-critical information includes all the control variables such as: switching frequency information, duty cycle information and all the sensor information. The provision for non-critical data transfer is designed to support tasks such as initialization and software reconfiguration of the hardware managers. Non-critical data transfer is allowed only after all the time critical data is exchanged.

Three types of time-critical data frames:

The control data frame, the synchronization frame and the command frame.

The data frame consists of a command indicating the beginning of the data packet, the address of the node, the data field and an error check. The data field's configuration depends on the particular application and type of hardware manager. In a ring network, each node introduces a delay in the data propagation path. This means that if a synchronization command is sent through the network, each node will receive the command with as many time delays T_d as there are nodes between that node and the master node. The time delay T_d in the hardware test-bed is typically around 460 ns. This means that the error in synchronization will generate time shifted PWM signals at the outputs, causing low-frequency harmonics. This problem is solved with the synchronization sequence.

The synchronization frame starts with the synchronization command, and is followed by 8 bit long data blocks containing addresses of slave nodes and filler

fields, for which it takes T_d to be transmitted and are used for propagation delay compensation. The first address to be transmitted is of the slave node that is last to receive the frame. The number of address data blocks sent equals the number of slave nodes on the ring, which need to be synchronized. The first field is a synchronization command that alerts the nodes to wait for their time to synchronize. Next are the address fields of the nodes being synchronized. After the synchronization command is passed, the node awaits its address field. When the address is received, the node generates the synchronization signal. Because all the addresses are in reverse order and time delayed for the node propagation delay, all the addresses will arrive at the destination nodes at almost the same time.

3.9 Conclusion

This chapter discussed a brief of communication protocols and Field buses viz, Ethernet bus, Mod bus, Profi bus used to design and develop the distributed controllers. Ethernet bus is preferred for high throughput and faster system response. Mod bus/Profi bus is preferred to connect the slow analog inputs to the system.

Chapter IV

Methodology of motion control

4.1 Introduction

Motion in industries is achieved using electric motors. In order to perform at desired operating points dynamically, these motors are required to control by power electronics devices which process the input electric energy to the motor. Such controlled motor is called electric drive. Electrical drives play an important role in the field of energy efficiency. Modern power electronic drives provide good opportunities to efficiently control the energy flows. These drives can be classified in two category, (i) DC drives, and (ii) AC drives. In this project , DC drive is controlled by 8051micro controller while ac i*nduction motor drive is controlled* by direct vector control method. This chapter describes the algorithm for DC machine drive and AC induction motor drives. Since these drives consume major electric energy and are commonly in use, the same has been considered for the developing the distributed controllers system. Some of the advantages of drives are as follows:

- Smoother operation
- Acceleration control
- Different operating speed for each process recipe
- Compensate for changing process variables
- Allow slow operation for setup purposes
- Adjust the rate of production
- Allow accurate positioning
- Control torque or tension
- Allow catching of spinning load (e.g., column of water) after outage.

4.2 DC drives

DC drives are DC motor speed control systems. Since the speed of a DC motor is directly proportional to armature voltage and inversely proportional to motor flux (which is a function of field current), either armature voltage or field current can be used to control speed. In DC, torque is controlled using the armature current and field current.

4.2.1 Algorithm for DC drive control

Figure 4.1 shows a model of separately excited DC motor. When a separately excited motor is excited by a field current of I_f and an armature current of I_a flows in the circuit, the motor develops a back EMF and a torque to balance the load torque at a particular speed. The field current I_f is independent of a separately excited motor is independent of the armature current I_a. Any change in the armature current has no effect on the field current. The I_f is normally much less than the I_a. The relationship of the field and armature are shown in Equation.

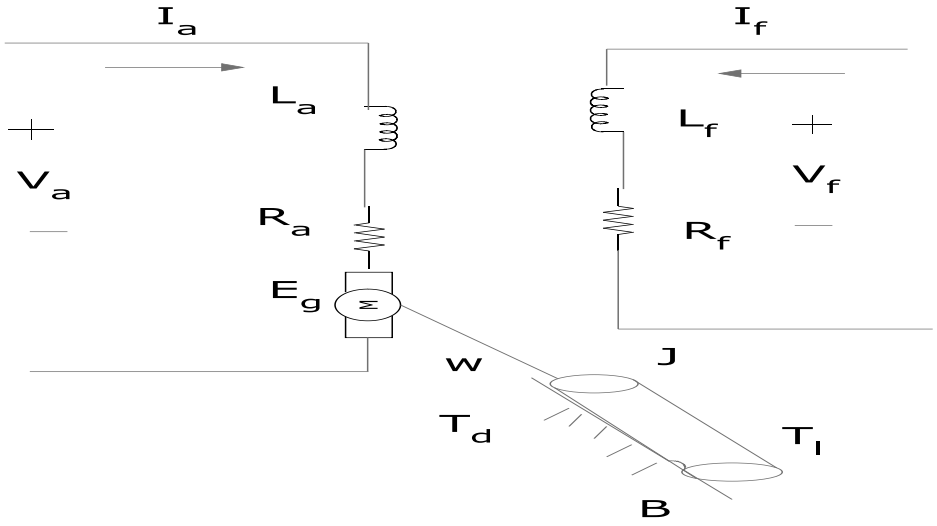

Figure 4.1 Equivalent circuit of separately excited DC motor

Instantaneous field current:

$$v_f = R_f i_f + L_f \frac{d i_f}{d t} \quad \textbf{(4.1)}$$

where R_f and L_f are the field resistor and inductor respectively.

Instantaneous Armature Current

$$v_a = R_a i_a + L_a \frac{d i_a}{d t} + e_g \quad \textbf{(4.2)}$$

where R_a and L_a are the armature resistor and inductor respectively.

The motor back EMF which is also known as speed voltage is expressed as

$$e_g = k_v w i_f \quad \textbf{(4.3)}$$

The torque developed by the motor is

$$T_d = k_t i_f i_a \quad \textbf{(4.4)}$$

where k_v is the back emf constant and k_t is motor torque constant (in V/A-rad/s) and w is the motor speed (rad/s).

In the armature controlled DC motor, the field current is kept constant, so that eqn. 4.4 can be written as

$$T_d = k_T i_a$$

For normal operation, the developed torque must be equal to the load torque plus the friction and inertia, i.e.:

$$T_d ¿ J \frac{d w}{d t} + T_L + B_w \quad \textbf{(4.5)}$$

The developed power $P_d = T_d w$

where B = viscous friction constant (N.m/rad/s)

T_L = load torque (N.m)

J = inertia of the motor ($kg.m^2$)

Under steady-state operations, a time derivative is zero. Assuming the motor is not saturated.

For field circuit,

$$V_f = I_f R_f \qquad \textbf{(4.6)}$$

The back EMF is given by:

$$E_g = K_v w I_f \qquad \textbf{(4.7)}$$

The armature circuit,

$$V_a = I_a R_a + E_g = I_a + K_v w I_f \qquad \textbf{(4.7)}$$

Thus, the motor speed can be easily derived:

$$w = (V_a - I_a R_a)/K_v I_f \qquad \textbf{(4.8)}$$

If the field current is kept constant, the speed motor speed depends on the supply voltage, armature current and hot resistance of armature. This algorithm has used in dc drive control with 8051 micro controller in next chapter. It is assumed that armature reaction in the motor and the voltage drops in the brushes have neglected.

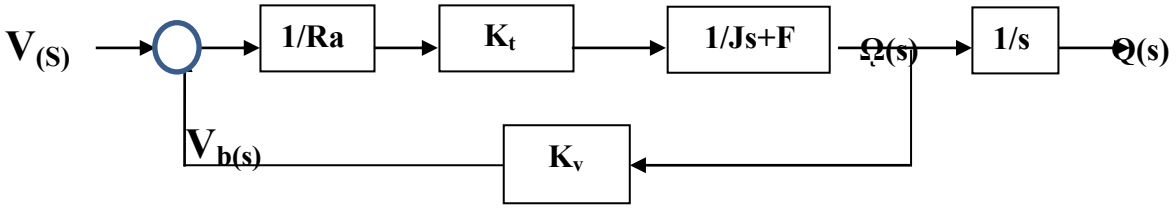

Fig. 4.2 Block dia. Of DC drive

4.3 Algorithm for AC drive:

There are many methods for controlling induction motor drive. In this project direct or feedback vector control method is used . Hence the algorithm of the same is presented in this section. First induction model is presented, then direct vector control algorithm is developed.

The vector control technique, which is also known as field – oriented control, allows a squirrel-cage induction motor to be driven with high dynamic performance. This technique decouples the two components of stator current: one providing the air-gap flux and the other producing the torque. It provides independent control of flux and torque. The stator currents are converted to a

fictitious synchronously rotating reference frame aligned with the flux vector and are transformed back to the stator frame before feeding back to the machine. The two components are d-axis i_{ds} analogous to armature current, and q-axis i_{qs} analogous to the field current. Thus induction motor can be modelled most simply using two quadrature currents rather than the familiar three phase currents actually applied to the motor. These two currents called direct (I_d) and quadrature (I_q) are responsible for producing flux and torque respectively in the motor. By definition, the I_q current is in phase with the stator flux, and I_d is at right angles. Of course, the actual voltages applied to the motor and the resulting currents are in the familiar three-phase system. The move between a stationary reference frame and a reference frame, which is rotating synchronous with the stator flux, becomes then the problem. This leads to the second fundamental idea behind vector control

In the indirect vector control method, the rotor field angle and thus the unit vectors are indirectly obtained by summation of the rotor speed and slip frequency.

4.3.1 Direct and Quadrature-Axis Transformation

The vector control technique uses the dynamic equivalent circuit of the induction motor. There are at least three fluxes (rotor, air gap and stator) and three currents or mmfs (in stator, rotor, and magnetizing) in an induction motor. For fast dynamic response, the interactions between currents, fluxes and speed, must be taken into account in obtaining the dynamic model of the motor and determining appropriate control strategies.

All fluxes rotate at synchronous speed. The three-phase currents create mmfs (stator and rotor), which also rotate at synchronous speed. Vector control aligns axes of an mmf and flux orthogonally at all times. It is easier to align the stator current mmf orthogonally to the rotor flux.

A symmetrical three phase induction machine with stationary reference frame (as-bs-cs) variables into two phase stationary reference frame (d^s-q^s) variable

and then transform these to synchronously rotating reference frame (d^e-q^e) and vice versa. Assume that the d^s-q^s axes are oriented at ø angle. The voltage Vds^s and Vqs^s can be resolved into as-bs-cs components and can be represented in the matrix form as

$$\begin{bmatrix} Vas \\ Vbs \\ Vcs \end{bmatrix} = \begin{bmatrix} Cos\emptyset & Sin\emptyset & 1 \\ Cos(\emptyset - 120) & Sin(\emptyset - 120) & 1 \\ Cos(\emptyset + 120) & Sin(\emptyset + 120) & 1 \end{bmatrix} \begin{bmatrix} Vqs^s \\ Vds^s \\ Vos^s \end{bmatrix} \quad \textbf{(4.9)}$$

The corresponding inverse relation is

$$\begin{bmatrix} Vqs^s \\ Vds^s \\ Vos^s \end{bmatrix} = 2/3 \begin{bmatrix} Cos\,(\emptyset) & Cos\,(\emptyset - 120) & Sin(\emptyset - 120) \\ Sin(\emptyset) & Cos(\emptyset + 120) & Sin(\emptyset + 120) \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} Vas \\ Vbs \\ Vcs \end{bmatrix} \textbf{(4.10)}$$

Where Vos^s is added the zero sequence component ,which may or may not be present. we have considered voltage as the variable the current and flux linkage can be transformed by similar equations .when d^e –q^e axes rotating synchronously which rotate as synchronous speed w_e with respect to the d^s-q^s axes and the angle $\theta_e = w_e t$ the two phase d^s-q^s windings are transformed into the hypothetical winding mounted on the d^e-q^e axes.the voltage on the d^s-q^s axes can be converted (or resolved) into the d^e-q^e frame as follows

$$V_{qs} = V_{qs^s} \cos\emptyset_e - v_{ds^s} \sin\emptyset_e \quad \textbf{(4.11)}$$

$$V_{ds} = V_{qs^s} \sin\emptyset_e + \_v_{ds^e} \cos\emptyset_e \quad \textbf{(4.12)}$$

Again resolving the rotating frame parameters into a stationary frame, the relations are

$$V_{qs^s} = V_{qs} \cos\emptyset_e + v_{ds} \sin\emptyset_e \quad \textbf{(4.13)}$$

$$V_{ds^s} = - V_{qs} \sin\emptyset_e + v_{ds} \cos\emptyset_e \quad \textbf{(4.14)}$$

In synchronously rotating reference frame for the two phase machine we need to represent both d_s-q_s and d_r-q_r circuits and their variables in a synchronously rotating d_e-q_e frame .we can write the following stator circuits equation

$$V_{qs}{}^{s} = R_s i_{qs}{}^{s} + \frac{d}{dt}\Psi_{qs}{}^{s} \quad \textbf{(4.15)}$$

$$V_{ds}{}^{s} = R_s i_{ds}{}^{s} + \frac{d}{dt}\Psi_{ds}{}^{s} \quad \textbf{(4.16)}$$

where $\Psi_{qs}{}^{s}$ and $\Psi_{ds}{}^{s}$ are q axis and d axis stator flux linkages respectively when these equations are converted to d^e-q^e frame the following equations can be written

$$V_{qs} = R_s i_{qs} + \frac{d}{dt}\Psi_{qs} + w_e\Psi_{ds} \quad \textbf{(4.17)}$$

$$V_{ds} = R_s i_{ds} + \frac{d}{dt}\Psi_{ds} - w_e\Psi_{qs} \quad \textbf{(4.18)}$$

Where all the variables are in rotating form , the last term in equations can be defined as speed emf due to rotations of that axes that is when $w_e = 0$ the equations revert to stationary form .The flux linkage in the d^e and q^e axes induce emf in the q^e and d^e axes respectively, with $\pi/2$ lead angle .

If the rotor is not moving, that is, $w_r = 0$ the rotor equations for a doubly fed wound rotor machine will be similar to equations

$$V_{qr} = R_r i_{qr} + \frac{d}{dt}\Psi_{qr} + w_e\Psi_{dr} \quad \textbf{(4.19)}$$

$$V_{dr} = R_r i_{dr} + \frac{d}{dt}\Psi_{dr} - w_e\Psi_{qr} \quad \textbf{(4.20)}$$

Where all the variables and parameters are referred to the stator .Since the rotor actually moves at speed w_r the d-q axes fixed on the rotor move at a speed w_e-w_r equations should be modified as

$$V_{qr} = R_r i_{qr} + \frac{d}{dt}\Psi_{qr} + (w_e - w_r)\Psi_{dr} \quad \textbf{(4.21)}$$

$$V_{dr} = R_r i_{dr} + \frac{d}{dt}\Psi_{dr} - (w_e - w_r)\Psi_{qr} \quad \textbf{(4.22)}$$

The flux linkage expression in terms of the currents can be written as follows

$$\Psi_{qs} = L_{ls} i_{qs} + L_m(i_{qs} + i_{qr}) \quad \textbf{(4.23)}$$

$$\Psi_{qr} = L_{lr} i_{qr} + L_m(i_{qs} + i_{qr}) \quad \textbf{(4.24)}$$

$$\Psi_{qm} = L_m(i_{qs} + i_{qr}) \quad \textbf{(4.25)}$$

$$\Psi_{ds} = L_{ls} i_{ds} + L_m(i_{ds} + i_{dr}) \quad \textbf{(4.26)}$$

$$\Psi_{dr} = L_{lr} i_{dr} + L_m(i_{ds} + i_{dr}) \quad \textbf{(4.27)}$$

$$\Psi_{dm} = L_m(i_{ds} + i_{dr}) \quad \textbf{(4.28)}$$

Combining the above expressions with equations the electrical transient model in the terms of voltage and currents can be given in matrix form as

$$\begin{bmatrix} V_{qs} \\ V_{ds} \\ V_{qr} \\ V_{qr} \end{bmatrix} = \begin{bmatrix} R_s + SL_m & \omega_e L_s & SL_m & \omega_e L_m \\ -\omega_e L_s & R_s + SL_s & -\omega_e L_m & SL_m \\ SL_m & (\omega_e - \omega_r)L_m & R_r + SL_r & (\omega_e - \omega_r)L_r \\ -(\omega_e - \omega_r)L_m & SL_m & -(\omega_e - \omega_r)L_r & R_r + SL_r \end{bmatrix} \begin{bmatrix} i_{qs} \\ i_{ds} \\ i_{qr} \\ i_{qr} \end{bmatrix} \quad (4.29)$$

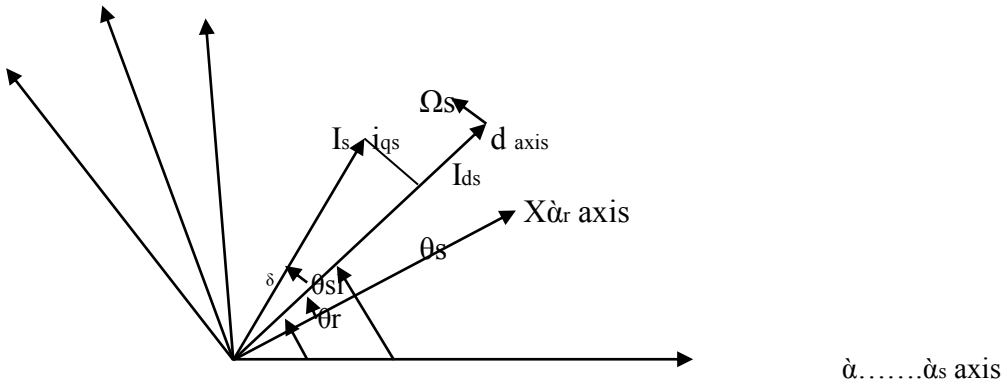

Fig 4.9: Axis of rotation for various quantities.

Where ω_s is the speed of reference frame (or synchronous speed), ω_m is rotor speed and

$$L_s = L_{ls} + L_m, L_r = L_{lr} + L_m \quad \textbf{(4.30)}$$

Subscripts l and m stand for leakage and magnetizing, respectively . The dynamic equivalent circuits of the motor in this reference frame are shown in figure:

$$\psi_{qs} = L_{ls} i_{qs} + L_m(i_{qs} + i_{qr}) = L_s i_{qs} + L_m i_{qr} \quad \textbf{(4.31)}$$

$$\psi_{ds} = L_{ls} i_{ds} + L_m(i_{ds} + i_{dr}) = L_s i_{ds} + L_m i_{dr} \quad \textbf{(4.32)}$$

$$\psi\hat{}_s = \sqrt{\psi_{qs}^2 + \psi_{ds}^2} \quad \textbf{(4.33)}$$

The rotor flux linkages are given by

$$\psi_{qr} = L_{lr}i_{qr} + L_m(i_{qr} + i_{qr}) = L_r i_{qr} + L_m i_{qr} \quad \textbf{(4.34)}$$

$$\psi_{dr} = L_{lr}i_{dr} + L_m(i_{dr} + i_{dr}) = L_r i_{dr} + L_m i_{dr} \quad \textbf{(4.35)}$$

$$\psi\hat{}_r = \sqrt{\psi_{qr}^2 + \psi_{dr}^2} \quad \textbf{(4.36)}$$

The air gap flux linkages are given by

$$\psi_{mq} = L_m(i_{qs} + i_{qr}) \quad \textbf{(4.37)}$$

$$\Psi_{md} = L_m(i_{ds} + i_{dr}) \quad \textbf{(4.38)}$$

$$\psi\hat{}_m = \sqrt{\psi_{mqs}^2 + \psi_{mds}^2} \quad \textbf{(4.39)}$$

Therefore the torque developed by the motor is given by

$$T_d = \frac{3}{2}p(\psi_{ds}i_{qs} - \psi_{qs}i_{ds}) \quad \textbf{(4.40)}$$

Where p is the number of poles. From matrix equation (4.29) as given above the rotor voltages in d- and q- axis as

$$v_{qr} = 0 = L_m\frac{di_{qs}}{dt} + (w_s - w_m)L_m i_{ds} + (R_r + L_r)\frac{di_{qr}}{dt}(w_e - w_r)L_r i_{dr}$$

$$v_{dr} = 0 = L_m\frac{di_{ds}}{dt} + (w_s - w_m)L_m i_{qs} + (R_r + L_r)\frac{di_{dr}}{dt}(w_e - w_r)L_r i_{qr}$$

Which, after substituting Ψ_{qr} from eqn. (4.34) and Ψ_{dr} from eqn. (4.35) , give

$$\frac{d\Psi_{qr}}{dt} + (R_r i_{qr}) + (w_e - w_m)\Psi_{dr} = 0 \quad \textbf{(4.41)}$$

$$\frac{d\Psi_{dr}}{dt} + (R_r i_{dr}) + (w_e - w_m)\Psi_{qr} = 0 \quad \textbf{(4.42)}$$

Solving for i_{qr} from Eqn. (4.34) and i_{dr} from Eqn .(4.35) , gives

$$i_{qr} = \frac{1}{L_r}\Psi_{qr} - \frac{L_m}{L_r}i_{qs} \quad \textbf{(4.43)}$$

$$i_{dr} = \frac{1}{L_r}\Psi_{dr} - \frac{L_m}{L_r} i_{ds} \quad \textbf{(4.44)}$$

Substituting rotor currents i_{qr} and i_{dr} into Eqn . (4.41) and Eqn . (4.42) ,we get

$$\frac{d\Psi_{qr}}{dt} + \frac{L_r}{R_r}\Psi_{qr} - \frac{L_m}{L_r} R_r i_{qs} + (w_s - w_m)\Psi_{dr} = 0 \quad \textbf{(4.45)}$$

$$\frac{d\Psi_{qr}}{dt} + \frac{L_r}{R_r}\Psi_{qr} - \frac{L_m}{L_r} R_r i_{qs} + (w_s - w_m)\Psi_{dr} = 0 \quad \textbf{(4.46)}$$

To eliminate transients in the rotor flux and the coupling between the two axes, the following conditions must be satisfied

$$\Psi_{qr} = 0 \; and \; \Psi_r = \sqrt{\Psi_{dr}^2 + \Psi_{qr}^2} = \Psi_{dr} \quad \textbf{(4.47)}$$

Also, the rotor flux should remain constant so that

$$\frac{d\Psi_{dr}}{dt} = d\frac{\Psi_{qr}}{dt} = 0 \quad \textbf{(4.48)}$$

With conditions in Eqn. (4.47) and Eqn. (4.48), the rotor flux Ψ_r is aligned on the d^e axis and we get

$$w_s - w_m = w_{sl} = \frac{L_m R_r}{\Psi_r L_r} i_{qs} \quad \textbf{(4.49)}$$

$$\frac{L_r}{R_r}\frac{d\Psi_r}{dt} + \Psi_r = L_m i_{ds} \quad \textbf{(4.50)}$$

Substituting the expressions for i_{qr} from Eqn. (4.43) into Eqn. (4.34) and i_{dr} from Eqn. (4.44) into Eqn. (4.35) , we get

$$\Psi_{qs} = (L_s - \frac{L_m^2}{L_r}) i_{qs} + \frac{L_m}{L_r}\Psi_{qr} \quad \textbf{(4.51)}$$

$$\Psi_{ds} = (L_s - L_M^2) i_{ds} + \frac{L_m}{L_r}\Psi_{dr} \quad \textbf{(4.52)}$$

Substituting above equations (4.51) and (4.52) into Eqn. (4.40) gives the developed torque as

$$T_d = \frac{3p}{2}\frac{L_m}{L_r}\left(\Psi_{dr} i_{qs} - \Psi_{qr} i_{qs}\right) = \frac{3p}{2}\frac{L_m}{L_r}\Psi_r i_{qs} \quad \textbf{(4.53)}$$

If the rotor flux Ψ_r, remains constant, becomes

$$\Psi_r = L_m i_{ds} \quad \textbf{(4.54)}$$

Which indicates that the rotor flux is directly proportional to current i_{ds}. Thus T_d becomes

$$T_d = \frac{3p}{2}\frac{{L_m}^2}{L_r} i_{ds} i_{qs} = K_m i_{ds} i_{qs} \qquad \textbf{(4.55)}$$

where $K_m = \frac{3pL_m^2}{2L_r}$

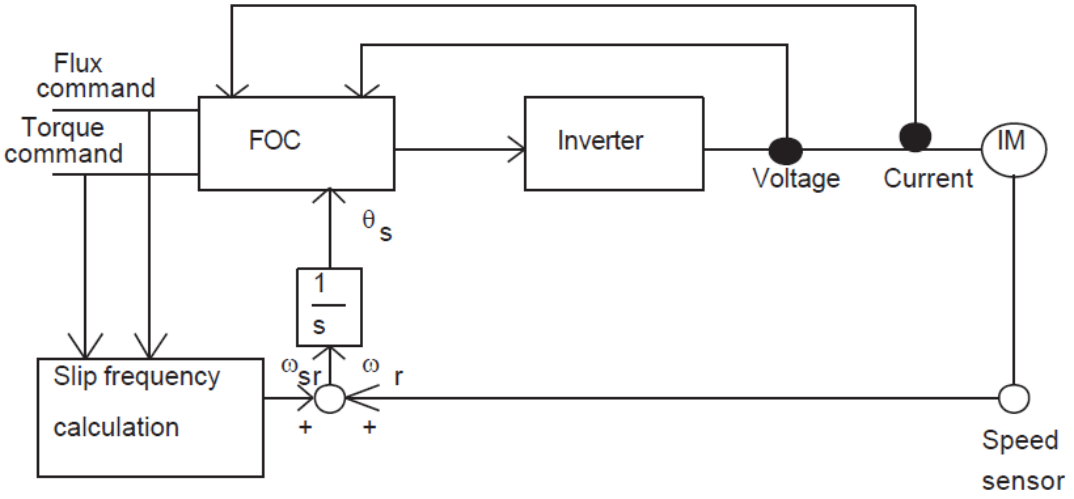

Fig 4.6: Block dia. of indirect vector control method

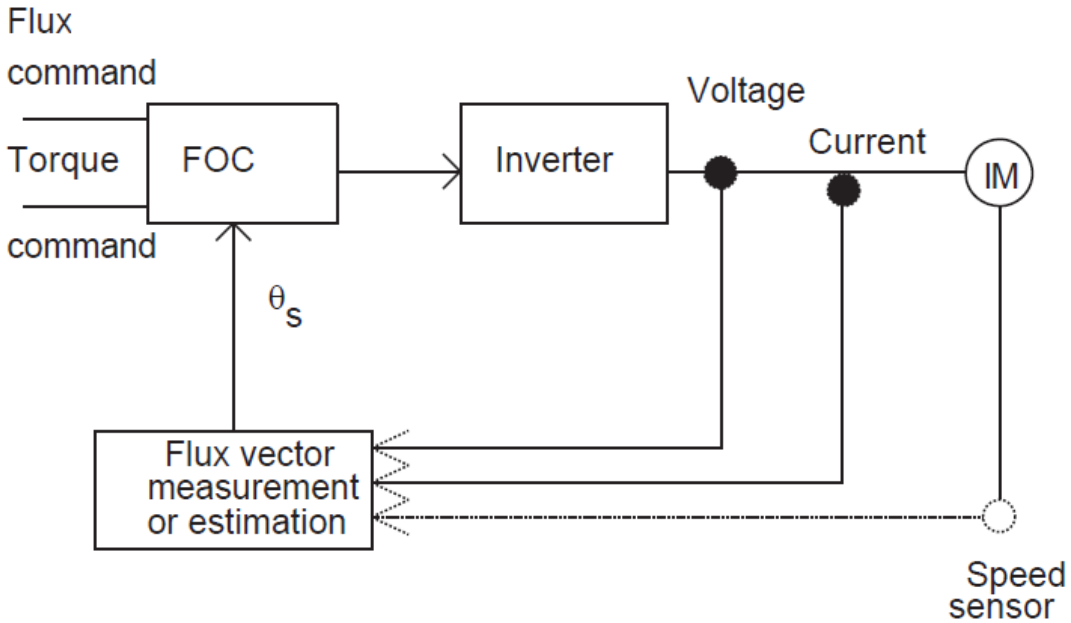

Fig 4.7: Block dia. Of direct vector control method

4.3.2 Direct torque control

The most modern technique is direct torque and stator flux vector control method (DTC). It has been realised in an industrial way by ABB, by using the theoretical background proposed by Blashke and Depenbrock during 1971-1985. This solution is based both on field oriented control (FOC) as well as on the direct self-control theory.

Direct torque control (DTC) is an induction motor control technique that has been successful because it explicitly considers the inverter stage and uses few machine parameters, while being more robust to parameter uncertainty than field-oriented control (FOC). The formal derivation of DTC based on singular perturbation and nonlinear control tools. The derivation elaborates an explicit relationship between DTC performance and machine characteristics; low-leakage machines are expected to perform better under DTC. It is shown that DTC is a special case of a sliding-mode controller based on the multiple time-scale properties of the induction machine. The known troublesome machine operating regimes are predicted and justified. Explicit conditions to guarantee stability are presented. DTC is shown to be a suboptimal controller because it uses more control effort than is required for flux regulation. Finally, compensation strategies that extend DTC are discussed. The derivation does not require space vector concepts thus, it is established that the traditional link between DTC and space vectors is not fundamental.

Starting with a few basics in a variable speed drive the basic function is to control the flow of energy from the mains to a process via the shaft of a motor. Two physical quantities describe the state of the shaft: torque and speed. Controlling the flow of energy depends on controlling these 15 quantities. In practice either one of them is controlled and we speak of "torque control" or "speed control". When a variable speed drive operates in torque control mode the speed is determined by the load. Torque is a function of the actual current

and actual flux in the machine. Likewise when operated in speed control the torque is determined by the load. Variable speed drives are used in all industries to control precisely the speed of electric motors driving loads ranging from pumps and fans to complex drives on paper machines rolling mills cranes and similar drives.

The idea is that motor flux and torque are used as primary control variables which is contrary to the way in which traditional AC drives control input frequency and voltage, but is in principle similar to what is done with a DC drive, where it is much more straightforward to achieve. In contrast, traditional PWM and flux vector drives use output voltage and output frequency as the primary control variables but these need to be pulse width modulated before being applied to the motor. This modulator stage adds to the signal processing time and therefore limits the level of torque and speed response time possible from the PWM drive.

In contrast, by controlling motor torque directly, DTC provides dynamic speed accuracy equivalent to closed loop AC and DC systems and torque response times that are 10 times faster. It is also claimed that the DTC does not generate noise like that produced by conventional PWM AC drives. And the wider spectrum of noise means that amplitudes are lower which helps to control EMI and RFI emissions. The basic structure of direct torque and stator flux vector control is Presented in Fig 4.8.

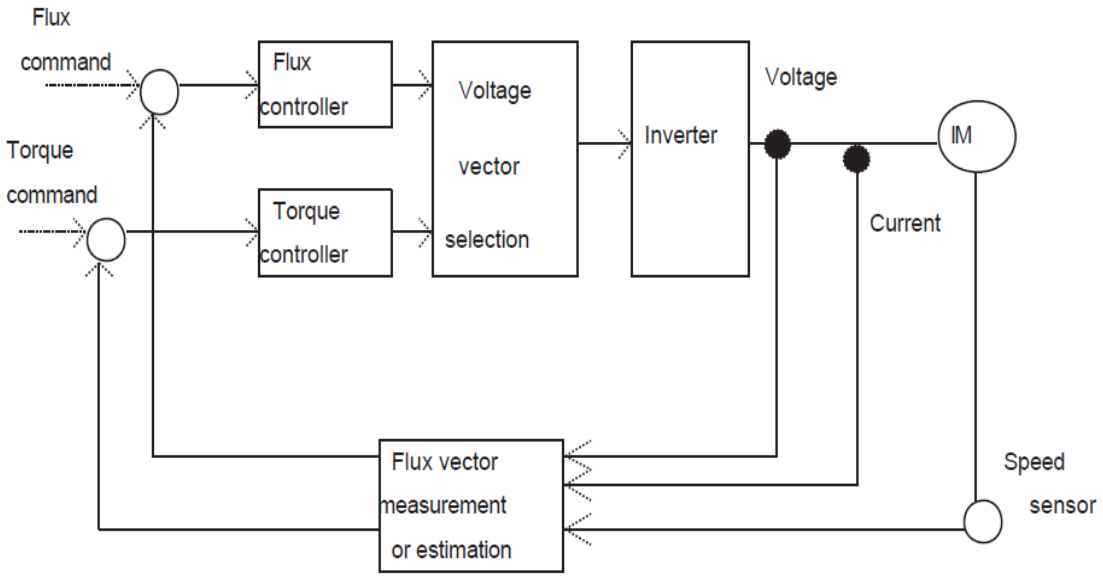

Fig. 4.8: Basic structure of direct torque and stator flux vector control

In DTC field orientation is achieved without feedback using advanced motor theory to calculate the motor torque directly and stator flux without using a modulator or a requirement for a tacho generator or position encoder to feed back the speed or position of the motor shaft. Both parameters are obtained instead from the motor itself. DTC's configuration also relies on two key developments - the latest high-speed signal processing technology and a highly advanced motor model precisely simulating the actual motor within the controller. A DSP (digital signal processor) is used together with ASIC hardware to determine the switching logic of the inverter.

The motor model is programmed with information about the motor, which enables it to determine parameters including stator resistance, mutual inductance saturation coefficients and motor inertia. The model also encompasses temperature compensation, which is essential for good static speed accuracy without encoder.

In normal operation, measurements of the two motor phase currents and the drive DC link voltage, together with information about the switching state of the inverter are fed into the motor model . The motor model then outputs control signals, which are accurate estimates of the actual motor torque and actual stator flux. All control signals are transmitted via optical links for high speed. In this way, the semiconductor switching devices of the inverter are supplied with an optimum switching pattern for reaching or maintaining an accurate motor torque. Also, both shaft speed and electrical frequency are calculated within the motor model. There is no need to feedback any shaft speed or position with tachometers or encoders to meet the demands of 95% of industrial applications. However, there will always be some special applications where even greater speed accuracy will be needed and when the use of an encoder improves the accuracy of speed control in DTC. But even then, the encoder does not need to be as costly or as accurate as the one used in traditional flux vector drives, as DTC only has to know the error in speed, not the rotor position.

The drive will have a torque response time typically better than 5ms. This compares with response for both flux vector PWM drives and DC drives fitted with encoders. The newer sensor less flux vector drives now being launched by other drives manufacturers have a torque response measured in hundreds of milliseconds.

DTC also provides exceptional torque control linearity. For the first time with an open loop AC drive, torque control can be obtained at low frequencies, including zero speed, where the nominal torque step can be increased in less than 1ms. The dynamic speed accuracy of DTC drives is better than any open loop AC drives and comparable to DC drives, which use feedback.

DTC brings other special functions, not previously available with AC drives, including automatic starting in all motor electromagnetic and mechanical states. There is no need for additional parameter adjustments, such as torque boost or starting mode selection, such as flying start. DTC control automatically adapts

itself to the required condition. In addition, based on exact and rapid control of the drive intermediate DC link voltage, DTC can withstand sudden load transients caused by the process, without any overvoltage or overcurrent trip.

4.4 Conclusion

This chapter discussed method used for controlling DC and induction drive. For controlling DC drive, armature voltage control method is used, where torque is developed by varying armature voltage. And for induction motor drive direct vector control method is used.

Chapter V

Implementation and testing

5.1 Introduction

The block diagram of Distributed controllers system of project is given below. In this project DC drive and Induction drive are controlled by two different Micro-controllers, 8051Microcontroller and AVR Microcontroller. In order to create distributed architecture the controllers connected to data highway bus(ether net bus). The technical specifications of the micro controller is also presented.

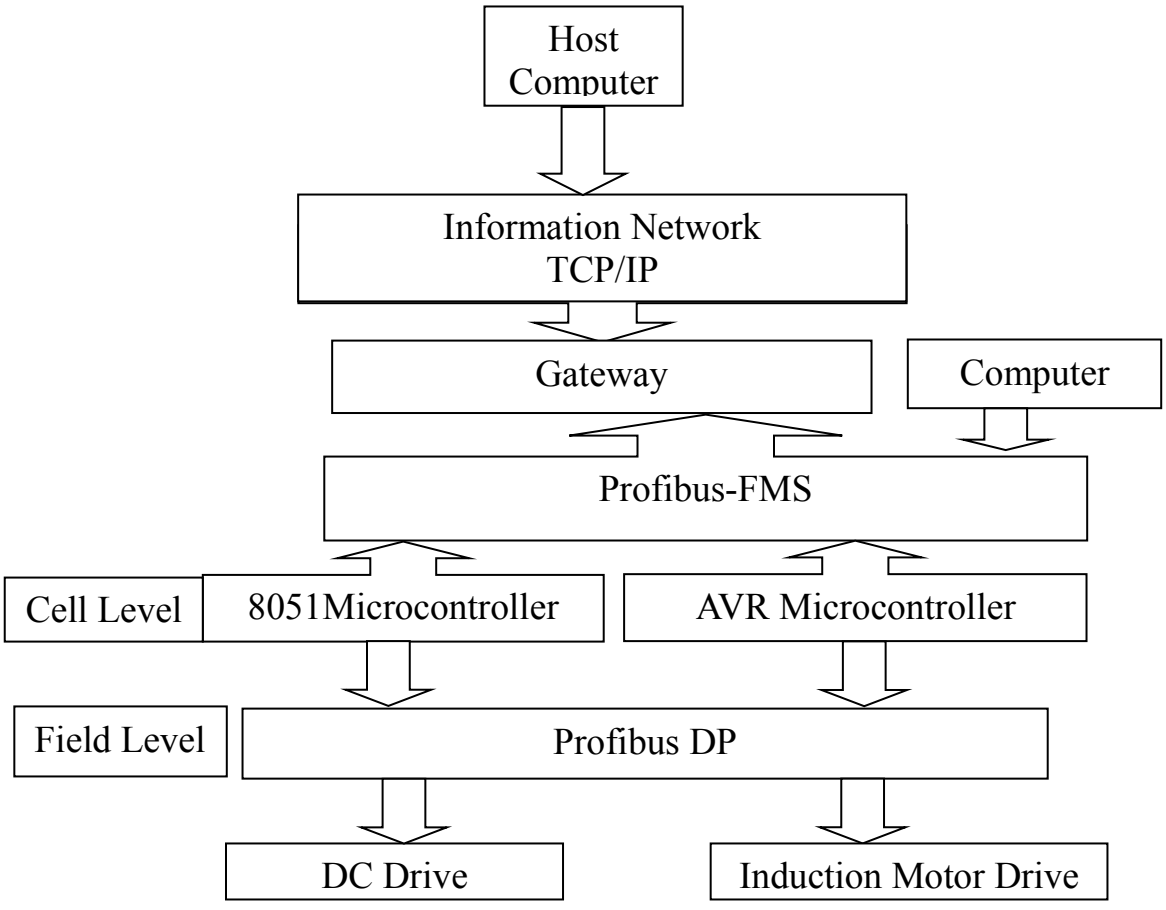

Fig 5.1: Block Diagram of DCSs.

5.2 8051 Microcontroller

The AT89C52 is a low-power, high-performance CMOS 8-bit microcomputer with 8K bytes of Flash programmable and erasable read only memory (PEROM). The device is manufactured using Atmel's high-density nonvolatile memory technology and is compatible with the industry-standard 80C51 and 80C52 instruction set and pin out.

The on-chip Flash allows the program memory to be reprogrammed in-system or by a conventional non-volatile memory programmer. By combining a versatile 8-bit CPU with Flash on a monolithic chip, the Atmel AT89C52 is a powerful microcomputer which provides a highly-flexible and cost-effective solution to many embedded control applications.

5.2.1 Pin configuration & Descriptions

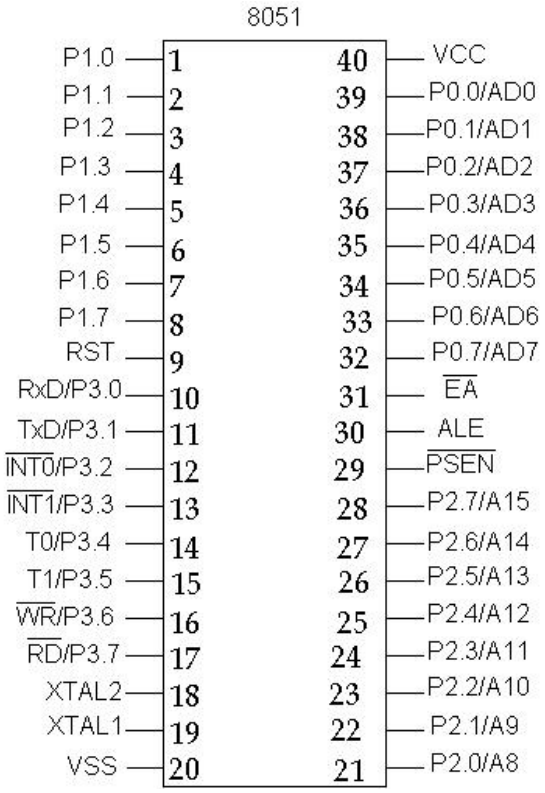

Fig. 5.2 Pin configuration of 8051

Pin-out description

Basic Pins

PIN 9: PIN 9 is the reset pin which is used to reset the microcontroller's internal registers and ports upon starting up. (Pin should be held high for 2 machine cycles). A logic one on this pin disables the microcontroller and clears the contents of most registers. In other words, the positive voltage on this pin resets the microcontroller. By applying logic zero to this pin, the program starts execution from the beginning.

PINS 18 & 19: The 8051 has a built-in oscillator amplifier hence we need to only connect a crystal at these pins to provide clock pulses to the circuit.

PINS 40 and 20: Pins 40 and 20 are VCC and ground respectively. The 8051 chip needs +5V 500mA to function properly, although there are lower powered versions like the Atmel 2051 which is a scaled down version of the 8051 which runs on +3V.

PINS 29, 30 & 31: This chip contains a built-in flash memory. In order to program this we need to supply a voltage of +12V at pin 31. If external memory is connected then PIN 31, also called EA/VPP, should be connected to ground to indicate the presence of external memory. PIN 30 is called ALE (address latch enable), which is used when multiple memory chips are connected to the controller and only one of them needs to be selected. PIN 29 is called PSEN. This is "program store enable". In order to use the external memory it is required to provide the low voltage (0) on both PSEN and EA pins.

There are 4 8-bit ports: P0, P1, P2 and P3.

PORT P1 (Pins 1 to 8): The port P1 is a general purpose input/output port which can be used for a variety of interfacing tasks. The other ports P0, P2 and P3 have dual roles or additional functions associated with them based upon the context of their usage. The port 1 output buffers can sink/source four TTL

inputs. When 1s are written to portn1 pins are pulled high by the internal pull-ups and can be used as inputs.

PORT P3 (Pins 10 to 17): PORT P3 acts as a normal IO port, but Port P3 has additional functions such as, serial transmit and receive pins, 2 external interrupt pins, 2 external counter inputs, read and write pins for memory access.

Pin 10: RXD Serial asynchronous communication input or Serial synchronous communication output.

Pin 11: TXD Serial asynchronous communication output or Serial synchronous communication clock output.

Pin 14: T0 Counter 0 clock input.

Pin 15: T1 Counter 1 clock input.

Pin 16: WR Write to external (additional) RAM.

Pin 17: RD Read from external RAM.

PORT P2 (pins 21 to 28): PORT P2 can also be used as a general purpose 8 bit port when no external memory is present, but if external memory access is required then PORT P2 will act as an address bus in conjunction with PORT P0 to access external memory. PORT P2 acts as A8-A15, as can be seen from above figure.

PORT P0 (pins 32 to 39) PORT P0 can be used as a general purpose 8 bit port when no external memory is present, but if external memory access is required then PORT P0 acts as a multiplexed address and data bus that can be used to access external memory in conjunction with PORT P2. P0 acts as AD0-AD7.

PORT P10: asynchronous communication input or Serial synchronous communication output. Oscillator Circuits The 8051 requires an external oscillator circuit. The oscillator circuit usually runs around 12MHz, although the 8051 (depending on which specific model) is capable of running at a maximum of 40MHz. Each machine cycle in the 8051 is 12 clock cycles, giving an effective cycle rate at 1MHz (for a 12MHz clock) to 3.33MHz (for

the maximum 40MHz clock). The oscillator circuit generates the clock pulses so that all internal operations are synchronized.

5.2.2 Internal Architecture of 8051 Microcontroller

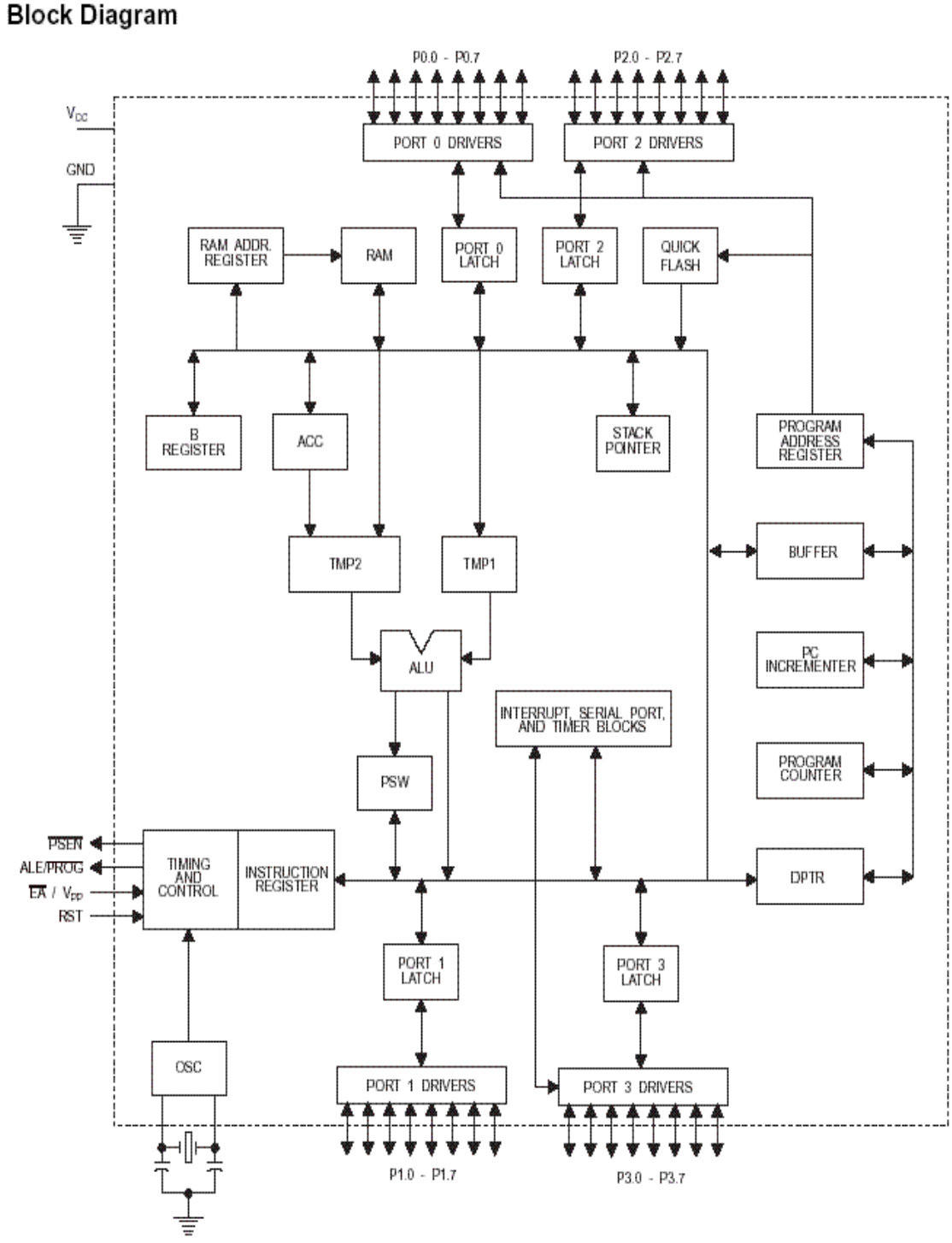

Fig. 5.3: Block dia. of 8051 Microcontroller

Data and Program Memory

The 8051 Microcontroller can be programmed in PL/M, 8051 Assembly, C and a number of other high-level languages. Many compilers even have support for compiling C++ for an 8051.
Program memory in the 8051 is read-only, while the data memory is considered to be read/write accessible. When stored on EEPROM or Flash, the program memory can be rewritten when the microcontroller is in the special programmer circuit.

Program Start Address

The 8051 starts executing program instructions from address 0000 in the program memory. The A register is located in the SFR memory location 0xE0. The A register works in a similar fashion to the AX register of x86 processors. The A register is called the accumulator, and by default it receives the result of all arithmetic operations.

Special Function Register

The Special Function Register (SFR) is the upper area of addressable memory, from address 0x80 to 0xFF. A, B, PSW, DPTR are called SFR. This area of memory cannot be used for data or program storage, but is instead a series of memory-mapped ports and registers. All port input and output can therefore be performed by memory mov operations on specified addresses in the SFR. Also, different status registers are mapped into the SFR, for use in checking the status of the 8051, and changing some operational parameters of the 8051.

General Purpose Registers

The 8051 has 4 selectable banks of 8 addressable 8-bit registers, R0 to R7. This means that there are essentially 32 available general purpose registers, although only 8 (one bank) can be directly accessed at a time. To access the other banks, we need to change the current bank number in the flag status register.

A and B Registers, The A register is located in the SFR memory location 0xE0. The A register works in a similar fashion to the AX register of x86 processors. The A register is called the accumulator, and by default it receives the result of all arithmetic operations. The B register is used in a similar manner, except that it can receive the extended answers from the multiply and divide operations. When not being used for multiplication and Division, the B register is available as an extra general-purpose register.

5.3 AVR Microcontroller

ATmega16 is an 8-bit high performance microcontroller of Atmel's Mega AVR family with low power consumption. Atmega16 is based on enhanced RISC (Reduced Instruction Set Computing) architecture with 131 powerful instructions. Most of the instructions execute in one machine cycle. Atmega16 can work on a maximum frequency of 16MHz. ATmega16 has 16 KB programmable flash memory, static RAM of 1 KB and EEPROM of 512 Bytes. The endurance cycle of flash memory and EEPROM is 10,000 and 100,000, respectively. ATmega16 is a 40 pin microcontroller. There are 32 I/O (input/output) lines which are divided into four 8-bit ports designated as PORTA, PORTB, PORTC and PORTD. ATmega16 has various in-built peripherals like USART, ADC, Analog Comparator, SPI, JTAG etc. Each I/O pin has an alternative task related to in-built peripherals.

5.3.1 Internal architecture of AVR Microcontroller

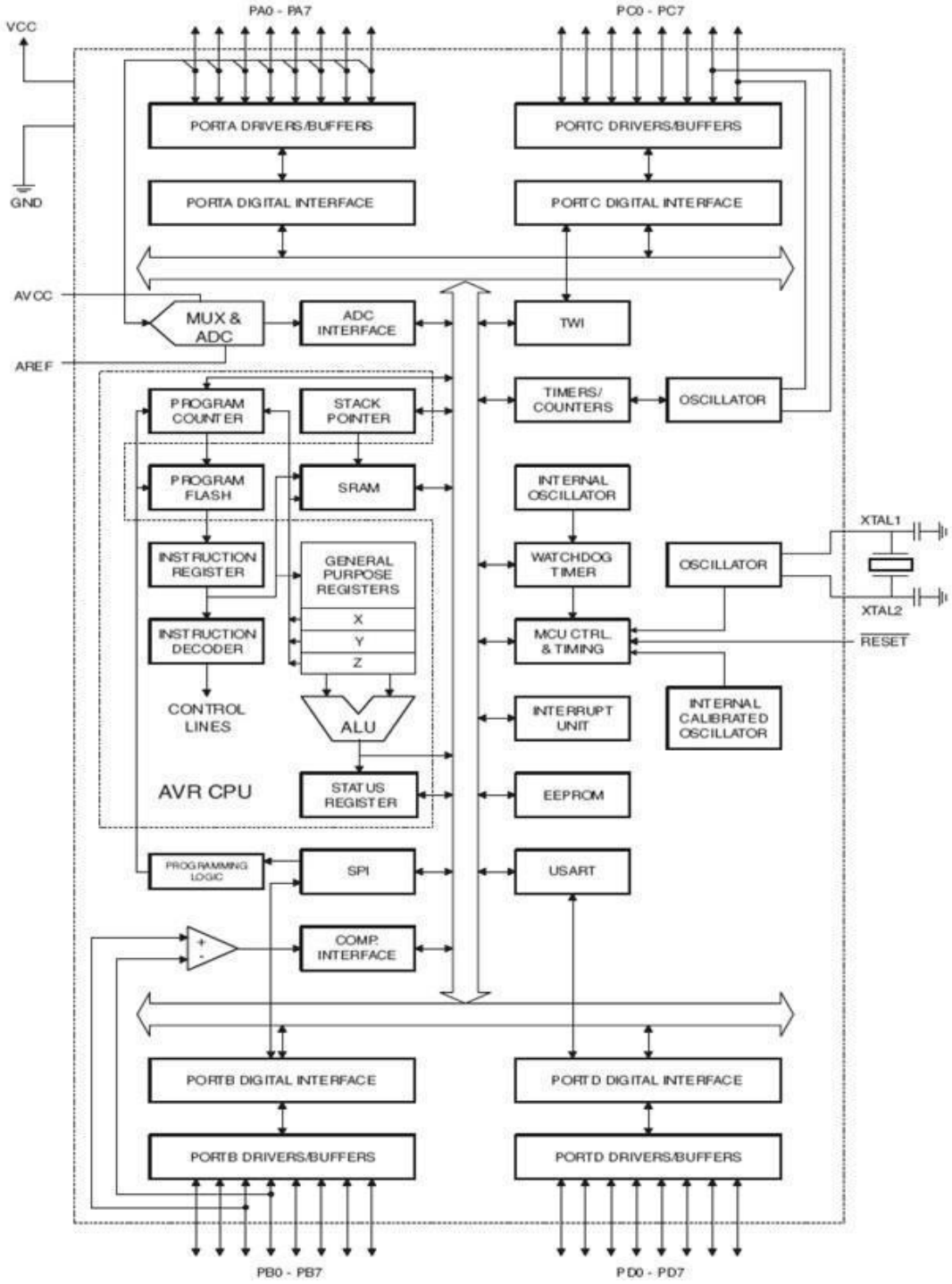

Fig 5.4 Block Diagram of AVR Microcontroller

5.3.2 Pin Configuration and description

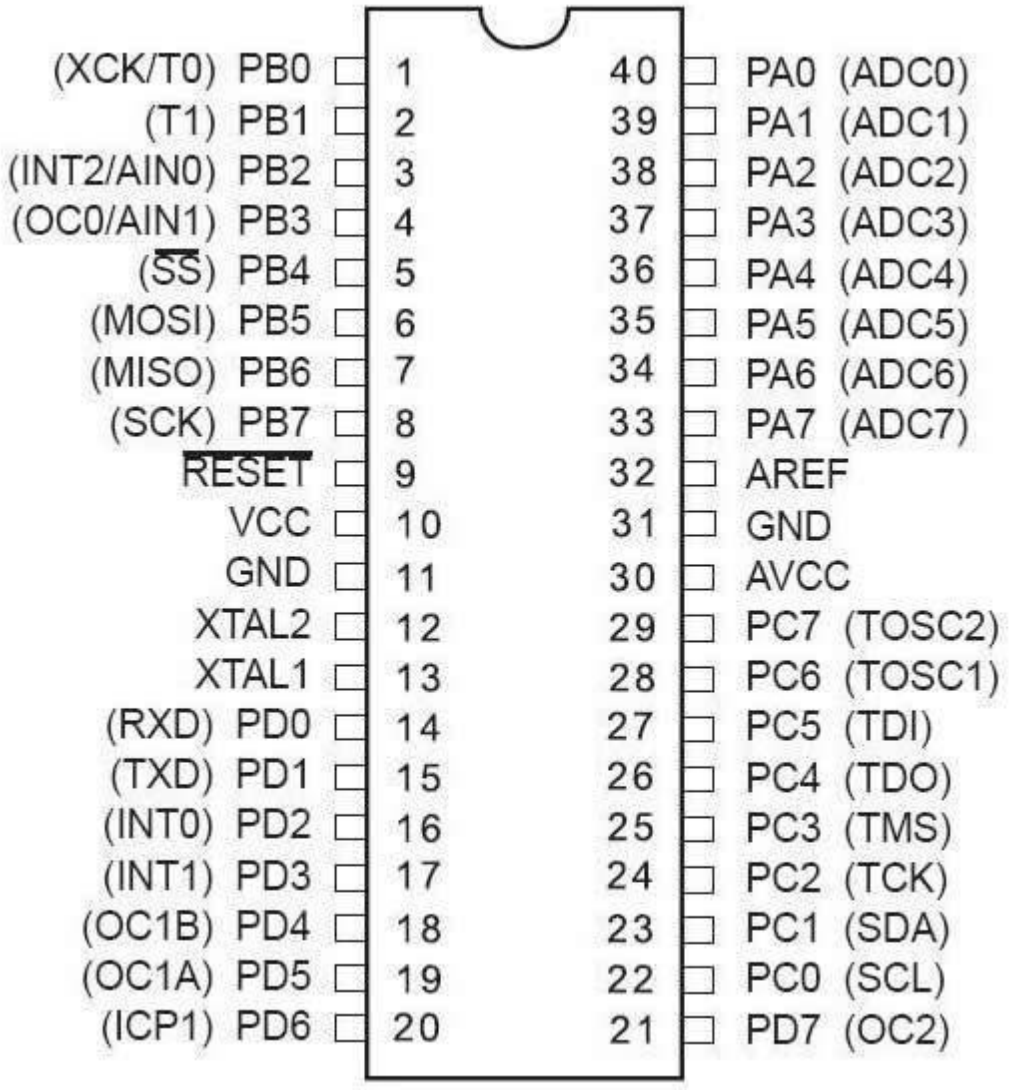

Fig. 5.5 Pin configuration of AVR Microcontroller

Pin out description

VCC: Digital supply voltage. (+5V)

GND: Ground. (0 V) Note there are 2 ground Pins.

Port A (PA7 - PA0) Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port B (PB7 - PB0) Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port B also serves the functions of various special features of the ATmega16 as listed on page 58 of datasheet.
Port C (PC7 - PC0) Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port C also serves the functions of the JTAG interface and other special features of the ATmega16 as listed on page 61 of datasheet. If the JTAG interface is enabled, the pull-up resistors on pins PC5 (TDI), PC3 (TMS) and PC2 (TCK) will be activated even if a reset occurs.
Port D (PD7 - PD0) Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port D also serves the functions of various special features of the ATmega16 as listed on page 63 of datasheet.
RESET: Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running
XTAL1: External oscillator pin 1
XTAL2: External oscillator pin 2
AVCC: AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter. AREF: AREF is the analog reference pin for the A/D Converter.

5.4 AVR Studio 4 Integrated Development Environment

AVR Studio is the Integrated Development Environment for developing 8-bit AVR applications in Windows NT/2000/XP/Vista/7 environments. “An integrated development environment (IDE) also known as integrated design environment or integrated debugging environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of:

- a source code editor,
- a compiler and/or an interpreter,
- build automation tools,
- a debugger.

AVR Studio 4 is a free IDE developed by ATMEL for writing and debugging AVR applications provides a complete set of features including debugger supporting run control including source and instruction-level stepping and breakpoints; registers, memory and I/O views; and target configuration and management as well as full programming support for standalone programmers.
To develop applications in C language is need the AVR-GCC- C compiler for AVR microcontrollers. “WinAVRTM (pronounced "whenever") is a suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. It includes the GNU GCC compiler for C and C++.”
AVR Studio 4 features:

- Integrated Development Environment
- Integrated Simulator
- Integrated Assembler
- Write, Compile and Debug
- Fully Symbolic Source-level Debugger
- Configurable Memory Views (SRAM/EEPROM/Flash/Registers and I/O)
- Extensive Program Flow Control Options
- Unlimited Number of Break Points
- Trace Buffer and Trigger Control
- Online HTML Help
- Variable Watch/Edit Window with Drag-and-Drop Function
- Simulator Port Activity Logging and Pin Input Stimuli

- File Parser Support: COFF/UBROF6/UBROF8 and Hex Files
- Language support: C, Pascal, BASIC, and Assembly

5.5 Distributed microcontroller System Application:

To demonstrate the utility of distributed microcontroller DC drive and induction motor drive are connected with 8051 microcontroller and AVR microcontroller respectively. The control software protocol are developed based on algorithm of chapter IV. The fig. 5.1 shows the distributed microcontroller application and architecture. The software protocol coding is presented in annexure I. The software protocol is developed in c language on computer system and downloaded to 8051 microcontroller. For induction motor drive software protocol is developed in AVR studio. The experiment are found satisfactory and in working mode for DC drive and induction motor drive both.

5.6 Conclusion

In this chapter Subsystem viz 8051 microcontroller and AVR microcontroller are described, including their internal features. The distributed control model for DC drive and induction motor drive has been developed and implemented.

Chapter VI
Conclusion and further scope

The aim of project was to design distributed controllers system, in which DC drive and induction drive are controlled. For designing of distributed controllers system Ethernet bus is used for communication system.

Because Ethernet bus is preferred for high throughput and faster system response. While the MOD bus / Profi bus is preferred to connect the Slow anolog inputs to the system. For that, armature control method is used for DC drive and direct vector control method is used for induction motor drive, 8051 microcontroller is for controlling DC drive and AVR microcontroller is used for controlling induction motor drive. And finally experiments are found satisfactory and in working mode for DC drive and induction motor drive both.

The further scope of this project is that it can be implemented for larger systems. Internal algorithm for such is Fuggy logic, neural network as such can be developed. This can also be used for PLC.

References

1.Electro-Craft Corporation. DC Motors, Speed Controls, Servo Systems. Pergamon Press,1977.
2.Kim, J.-M. and Sul, S.-K. (1997). Speed control of interior permanent magnet synchronous motor drive for the flux weakening operation. *IEEE Trans. Ind. Ap* 33(1):43–48.
3.Mohan, N., Undeland, T. M., and Robbins, W. B. (1995). *Power Electronics - Converters, Applications and Design*. John Wiley & Sons, 2nd edition.

4.Khambadkon, A. M. & Holtz, J. (1991). Vector-controlled induction motor drive with a self-commissioning scheme, *IEEE Transactions on Industrial Electronics*, vol. 38, no. 5, (March 1991), pp. 322-327, ISSN 0278-0046

5. Bose, B.K. 1986. "Power Electronics and Drives", Prentice-Hall, Englewood Cliffs, NewJersey
6. "WorldFIP Basics," http://www.worldfip.org.

7. "Profibus Technical Description," http://www.profibus.com, September 1999.

8. "MACRO: Motion And Control Ring Optical," http://www.macro.org, May 1998.

9. V. Vlatkovic and D. Borojevic, "Digital-Signal-Processor-Based Control of Three-Phase Space Vector Modulated Converters," *IEEE Transactions on Industrial Electronics*, June 1994, Vol. 41, No. 3, pp. 326-332.

10. E. Bassi, F. Benzi, L. Lusetti, G.S. Buja, "Communication Protocols for Electrical Drives," *IEEE IECON Proceedings, 1995*, Vol. 2, pp.706-711.

11. A. Crane and T. J. McCoy, "Electromagnetic Compatibility Design for a 19 MW PWM Motor Drive."

12. R. Lewis, "Design of distributed control systems in the next millennium", *Computing & Control Engineering Journal*, Volume: 8 Issue: 4, August 1997.

13. Goktas F., *Distributed Control of Systems over Communication Networks*", Ph.D Thesis, University of Pennsylvania, 2000
14. Chesney C., *Which Bus Architecture Is Best for You?* EE:-Evaluation Engineering, V 37 N9 1998
15.Slemon, G.R. 1989. "Modelling of Induction Machines for Electric Drives", in IEEE Trans.Ind. Appl., Vol. 25, No. 6, pp. 1126-1131
16. Slemon, G.R. 1994. "Electrical Machines for Variable-Frequency", in Proceedings of the IEEE, Vol. 82, No. 8, pp.1123-1138
17. Slemon, G.R. 1989. "Modelling of Induction Machines for Electric Drives", in IEEE Trans. Ind. Appl., Vol. 25, No. 6, pp. 1126-1131
18. Wikipedia - The free Encyclopaedia http://en.wikipedia.org/wiki/Modbus.
19. Wikipedia - The free Encyclopaedia http://en.wikipedia.org/wiki/Fieldbus
20. Wikipedia - The free Encyclopaedia http://en.wikipedia.org/wiki/Ethernetbus
21.Muhammad H .Rashid6. "Power Electronics ckt ,Devices, and Application", III edition, Pearson education, 2006.
22. Mazidi, Mazidi ,Mckinlay, IInd edition, Pearson education, 2006.
23. Electro-Craft Corporation. DC Motors, Speed Controls, Servo Systems. Pergamon Press,1977.
24. P.C. Krause. Analysis of electric machinery. McGraw Hill, 1986.

Annexure-I

Coding of 8051 microcontroller

```
#include<reg51.h>
#include <string.h>
sbit rs = P3^1;                         // rs pin of LCD
sbit en = P3^0;                         // en pin of LCD
sbit rw = P3^2;                         // rw pin of LCD
sbit b = P0^7;                          // busy flag
sbit led1=P2^0;                         // run indicator
sbit led2=P2^1;                         // stop indicator
sbit led3=P2^2;                         //    clockwise    direction
indicator
sbit led4=P2^3;                         //  anticlockwise  direction
indicator
sbit PWM=P2^4;                          // PWM output
sbit RL1=P2^5;                          // relay 1 pin
sbit RL2=P2^6;                          // relay 2 pin

unsigned int x=10;                      // ontime
unsigned int y=10;                      // offtime
unsigned int m=0;                   // mode
unsigned int d=0;                   // direction
unsigned int t=100;                     // time
unsigned int r=0;                   // run flag

void start(void);                   // function initilization
void mode(void);
void direction(void);
void incspeed(void);
void decspeed(void);
void inctime(void);
void dectime(void);
void time(unsigned int);
void delay(unsigned int);
void keydly(void);
void busy(void);

void writecmd(unsigned char a)          // send command to LCD
  {
      busy();                                   //   check   busy
flag
      rs = 0;                                   // select command
register
      rw = 0;                                   // write enable
      P0 = a;                                   //  send  byte  to
LCD
      en = 1;                                   //  apply  strobe
pulse
      en = 0;
  }
void writedata(unsigned char b)         // send data to LCD
  {
      busy();                                   //   check   busy
flag
      rs = 1;                                   //   select   data
register
      rw = 0;                                   // write enable
      P0 = b;                                   //  write  enable

      en = 1;                                   //  send  byte  to
LCD
      en = 0;                                   //  apply  strobe
pulse
  }
void busy()                                 // check busy flag of
LCD
```

```
  {
      en = 0;                                          //        disable
display
      P0 = 0xFF;                                  // P0 as input
      rs = 0;                                          // select command
register
      rw = 1;                                          // read enable
      while(b==1)                                 // if busy bit is 1
      {
       en=0;                                           // remain withine
loop
       en=1;
      }
      en=0;
  }
void writestr(unsigned char *s)           // send string message to LCD
 {
      unsigned char l,i;
      l = strlen(s);                                   // get length of string
      for(i=0;i<l;i++)
      {
            writedata(*s);                             //  till  the  length  of
string
            s++;                                       //  send  characters  one
by one
      }
 }
void start()                                           // start rotating motor
  {
      if(m==0)                                         //    for    m=0    start
continuous mode
        {
            RL1=0;                                           // switch on RL1
            r=1;                                       // set run flag
            P1=0xFF;                                   // send all 1's to P1
            while(P1==0xFF)                            //   till   no   key   is
pressed
              {
                  led1=1;                                    //  indication  on
run LED
                  PWM=1;                                     //    send    high
logic to PWM pin
                  delay(x);                            // on time delay
                  PWM=0;                                     //  now  send  low
logic to PWM
                  delay(y);                            // off time delay
              }
            }
            else if(m==1)                              //    for    m=1    start
reversible mode
              {
                  r=1;                                 // set run flag
                  P1=0xFF;                             // send all 1's to P1
                  while(P1==0xFF)                      //   till   no   key   is
pressed
                    {

                        led1=1;                              // run LED=1
                        led3=1;
                        led4=0;
                        PWM=1;                               //  send  high  on
PWM pin
                        RL2=1;                               //    select    one
direction
                        RL1=0;                               // switch on RL1
                        time(t);                       //   wait   for   desired
time
                        RL1=1;                               // switch off RL1
                        led1=0;                              // run LED=0;
```

```
                        time(20);               // wait for 1 sec
                        led1=1;                       //    again    run
LED=1
                        led3=0;
                        led4=1;
                        RL2=0;                        //  select  other
direction
                        RL1=0;                        // switch on RL1

                        time(t);                // wait for desire time
                        RL1=1;                        // switch off RL1
                        led1=0;                       // run LED=0
                        time(20);               // wait for 1 sec
                     }
                        PWM=0;

                   }
               else if(m==2)                    //    for    m=2    start
jogging mode
                  {
                        r=1;                    // reset run flag
                        P1=0xFF;                      //  send  all  1's
to P1
                    while(P1==0xFF)             //   till   no   key   is
pressed
                      {
                        led1=1;
                        PWM=1;                        //  send  high  on
PWM pin
                        RL1=0;                        // switch on RL1
                        time(t);                // wait for 1 sec
                        RL1=1;                        // switch off RL1
                        PWM=0;                        //  send  low  on
PWM pin
                        led1=0;
                        time(20);
                      }

                   }
  }
void direction()                          // alter the direction
  {
      keydly();                                 // key debounce delay
      d++;                                      // increment count
      if((d%2)==0)                              //  check  for  even  or
odd
        {
            led3=1;                                 //   indicate   on
LEDs
            led4=0;
            RL2=1;                                  //  switch  ON  /
OFF RL2
        }
      else
        {
            led3=0;
            led4=1;
            RL2=0;
        }
  }
 void mode()                                                //   change
mode of rotation
   {
      keydly();                                             //      key
debounce delay
      writecmd(0x80);                                       //  display
message on first line first column
      m++;                                                  //
increment count
```

```
      if(m==3) m=0;                                          // if it is
3 reset it
      if(m==0)
        { writestr("mode:continuous ");   // otherwise display mode
        time(15);
        }
      else if(m==1)
    {writestr("mode:reversible ");
         time(15);
      }
      else if(m==2)
      {writestr("mode:jogging    ");
      time(15);
   }
   }
 void decspeed()                                                //
increase speed
   {
            int z;
            keydly();
      // key debounce
            writecmd(0xC0);
      // select second line on LCD
            if(y<14)
      // if not max pulse width
              {
                  x--;
                  y++;
      // increase it convert it in to
                  z=y-5+0x30;
      // 1 to 10 scale and ASCII
                  writestr("speed: ");                          //
diaplay speed on LCD
                  writedata(z);
                  writestr("       ");
              }
              else if(y==14) writestr("min speed: 9  ");    //  if  max
width display message
      }
void incspeed()
            // increase speed
   {
            int w;
            keydly();
            writecmd(0xC0);
            // key debounce

            if(y>6)
            // if not minimum width
              {
                  x++;
                  y-- ;
            // decrease it
                  w=y-5+0x30;
            // do same as above
                  writestr("speed: ");
                  writedata(w);
                  writestr("       ");
              }
             else if(y==6) writestr("max speed: 1  ");            // if
min width display message
      }
void inctime()
            // increase time
  {
      int p;
      keydly();
            // key debounce delay
```

```
      writecmd(0xC0);

      if(t<180)
            // if not max time
        {
            t+=20;
                  // increase it by 1 sec
            p=t/20;
            p=p+0x30;
            // convert it in to ASCII
            writestr("time: ");
            // display it
            writedata(p);
            writestr(" sec    ");
        }
      else if(t==180) writestr("max time: 9 sec");          //  if  max
time display message
  }
void dectime()
            // decrease time
  {
      int q;
      keydly();
            // key debounce delay
      writecmd(0xC0);
      if(t>20)
            // if not min time
        {
            t-=20;
                  // decrease it
            q=t/20;
            q=q+0x30;
            // do same as above
            writestr("time: ");
            writedata(q);
            writestr(" sec    ");
        }
      else if(t==20) writestr("min time: 1 sec");                 // if
min time display message
  }
void keydly()                                       // key debounce delay
  {
      int a,b;
      for(a=0;a<50;a++)
        for(b=0;b<1000;b++);
  }
void time(unsigned int c)                 // change time in seconds
 {
      int k;
      TL1 = 0xAF;                                   // use timer 1
      TH1 = 0x3C;                                   //  to  generate  50  ms
delay
      TR1 = 1;                                      // start timer
      for(k=0;k<=c;k++)                   //  rotate  loop  in  multiples
of 20
        {
            while(TF1==0);                          //   wait   till   timer
overflow
            TF1 = 0;                                // reset the flag
            TL1 = 0xAF;                             // reload it
            TH1 = 0x3C;
        }
      TR1 = 0;                                      // stop timer
  }
void delay(unsigned int c1)               //   change   time   in   micro
seconds
 {
      int a;
      TH0=0x9B;                                     // select timer 0
```

```
      TL0=0x9B;                                   //   to   generate   100
micro second delay
      TR0=1;                                            //   start   timer

      for(a=0;a<c1;a++)                     //  rotate  loop  between  5  to
15
        {
            while(TF0==0);                          //   wait   until   timer
overflow
            TF0=0;                                      // reset the flag
        }
      TR0=0;                                            // stop timer
  }

void main()
  {
      TMOD=0x12;                                    //  timer1  in  16  bit
timer, timer 0 in 8 bit auto reload mode
      P2=0xE0;                                      // LEDs off, relays OFF
      P0=0x00;                                      // P0, P3 output ports
      P3=0x00;
      writecmd(0x3C);                               // initilize LCD
      writecmd(0x0E);
      writecmd(0x01);
      writecmd(0x84);                               // display message
    writestr("DC Motor");                   // DC motor controller in
      writecmd(0xC3);                               // center of LCD
      writestr("Controller");
agin:P1=0xFF;                                       // P1 as input port
      while(P1==0xFF);                      // wait until any key press
loop:switch(P1)
        {
            case 0xFE:                              // for first key
                  keydly();                         // key debounce
                  writecmd(0x01);
                  writestr("motor start");
                  time(50);                                 // wait for
2.5 sec
                  writecmd(0x80);
            writestr("mode:continuous "); //  display  current  mode  and
speed
                  writecmd(0xC0);
                  writestr("speed: 5       ");
                  led1=1;                               // Run LED ON
                  led2=0;                               // stop LED OFF
                  led3=1;                               //      clockwise
direction ON
                  led4=0;                               //  anticlockwise
direction OFF
                  start();                          // sart rotating motor
                  break;
            case 0xFD:                              // for second key
                  keydly();                         // key debounce
                  r=0;                              // run flag reset
                  writecmd(0x01);
                  writestr("motor stop"); // display message
                  led1=0;                               // Run OFF
                  led2=1;                               // stop LED ON
                  led3=0;                               //      clockwise
direction OFF
                  led4=0;                               //  anticlockwise
direction OFF
                  PWM=0;                                //  low  logic  to
PWM pin
                  RL1=1;                                // relay1 off
                  break;
            case 0xFB:                              // for third key
```

```
                  mode();                              //   select   mode

                  if(r==1) start();       // jump to start if run flag
is set
                  break;
            case 0xF7:                              // for fourth key

                  direction();                        //   change   direction

                  if(r==1) start();       // jump to start if run flag
is set
                  break;
            case 0xEF:                              // for fifth key

                  incspeed();                         // increase speed
                  if(r==1) start();       // jump to start if run flag
is set
                  break;
            case 0xDF:                              // for sixth key

                  decspeed();                         // decrease speed

                  if(r==1) start();       // jump to start if run flag
is set
                  break;
            case 0xBF:                              //   for   seventh   key

                  inctime();                          // increase time
                  if(r==1) start();       // jump to start if run flag is
set
                  break;
            case 0x7F:                              // for eigth key
                  dectime();                          // decrease time

                  if(r==1) start();       // jump to start if run flag is
set
                  break;
         }
      if(r==1) goto loop;                             // if run flag is set jump
of key detect
      else goto agin;                                 // if not jump to again
  }
```

Annexure-II

coding of AVR microcontroller

Below is presented the main code of embedded application. Other files: USART.c, PI.c, and TIMERS.c are included in appendix.

```
#include <avr/io.h>
#include <stdlib.h>
#include <stdbool.h>
#include <avr/interrupt.h>
#include "USART.c"
#include "PI.c"
#include "TIMERS.c"
#define F_CPU 8000000UL
//regulation
const char cEncoderResolution=62; //resolution of encoder's rotary disc
volatile int i=0;
volatile int iMotorNumber=0;
volatile char cNumberOfIterations=2;
volatile char cSamplesOfSpeed[10];
volatile char cScalingFactor;
//motors speed & voltage
```

```
volatile int iPresetRPM[2];
volatile char cActualSpeed[2];
volatile int iActualRPM[2];
volatile int iSaturation[2];
//flags
volatile bool bFlagRegulation=false;
volatile bool bFlagPidIsOn=true;
volatile bool bFlagToSend=false;
volatile bool bFlagTransmit=true;
volatile bool bFlagReadyToSplit=false;
volatile bool bFlagSendInfo=false;
//transmission
volatile char cReceivedBuffer[11];
volatile char cReadChars=0;
volatile int iSumOfSpeed=0;
volatile char cDelay=0;
volatile char cTransmitionDelay=5;
//statistics
volatile unsigned long int iFramesError=0;
volatile unsigned long int iFramesRecived=0;
volatile unsigned long int iFramesTransmitted=0;
volatile unsigned long int iWorkTime=0;
//-----------------------------------------------------TIMER speed regulation is enabled
ISR(TIMER2_COMP_vect){//this interrupt is triggered after an appropriate time interval
cli();
if(bFlagTransmit==true){
if(cDelay>cTransmitionDelay){  //delaying transmission of cTransmitionDelay time intervals
bFlagToSend=true; //enabled transmission
cDelay=0;
}
if(i>cNumberOfIterations){
cActualSpeed[iMotorNumber]=(char)(iSumOfSpeed/(float)(i+1)); //computing mean value of the speed for i iterations
iActualRPM[iMotorNumber]=(int)cScalingFactor*cActualSpeed[iMotorNumber];   //scaling the speed mean value to RPM
bFlagRegulation=true; //enabled speed regulation
i=0;
iSumOfSpeed=0;
}
else{
cSamplesOfSpeed[i]=TCNT0; //inserting to an array values of speed in iteration
iSumOfSpeed+=TCNT0; //summing speed values
TCNT0 = 0x00; //counter reset
i++;
}
```

```
}
cDelay++;
iWorkTime++;
sei();
}
//------------------------------------------------------TIMER
//------------------------------------------------------USART RX
ISR(USART_RXC_vect){//this interrupt is triggered when USART receive character
//cli();
char c=UDR; //obtaining character from the USART
char cSumBuffor[3]={0,0,0};
int iChSumRead=0;
int iChSumCalc=0;
if(c==13){ //13 is ascii code of EOF(end of line)
for(int n=cReadChars-3;n<cReadChars;n++) cSumBuffor[n-
cReadChars+3]=cReceivedBuffer[n]; //retrieve checksum from received data
iChSumRead=atoi(cSumBuffor);
for(int n=0;n<cReadChars-4;n++) iChSumCalc+=(int)cReceivedBuffer[n]; //calculating
checksum of received packet
if(iChSumRead==iChSumCalc){ //corrected frame
bFlagReadyToSplit=true; //enabled string processing
iFramesRecived++;
iChSumCalc=0;
}
else{ //corrupted frame
iFramesError++;
cReadChars=0;
iChSumCalc=0;
for(int n=0;n<sizeof(cReceivedBuffer);n++) cReceivedBuffer[n]=0; //reset of receive
buffer
}
}
else{ //if character is not EOF, add it to the receive buffer
cReceivedBuffer[cReadChars]=c;
cReadChars++;
}
if(cReadChars>sizeof(cReceivedBuffer)) cReadChars=0;
//sei();
}
//------------------------------------------------------USART RX
void Regulation(void){
int iPI=0;
switch(iMotorNumber){ //choose number of regulated motor
case 0:
Timer0_init(); //start count pulses
if(bFlagPidIsOn==true){
if(iPresetRPM[0]==0) OCR1A=0; //if motor speed is 0, stop motor without regulator
else{
iPI=PI(i,(char)(iPresetRPM[0]/(float)cScalingFactor),cActualSpeed[0],cSamplesOfSpee
d); //obtain corrected value of PWM duty cycle to fulfill preset speed
if(OCR1A+iPI>1023) OCR1A=1023;
else OCR1A+=iPI;
}
}
Timer0_stop(); //stop count pulses
iMotorNumber=1;
PORTD |= (1<<PORTD7); //change input signal for the multiplexer, OC0 count pulses
for motor1
```

```
case 1:
Timer0_init(); //start count pulses
if(bFlagPidIsOn==true){
if(iPresetRPM[1]==0) OCR1B=0; //if motor speed is 0, stop motor without regulator
else{
iPI=PI(i,(char)(iPresetRPM[1]/(float)cScalingFactor),cActualSpeed[1],cSamplesOfSpee
d); //obtain corrected value of PWM duty cycle to fulfill preset speed
if(OCR1B+iPI>1023) OCR1B=1023;
else OCR1B+=iPI;
}
}
Timer0_stop(); //stop count pulses
iMotorNumber=0;
PORTD &= ~(1<<PORTD7); //change input signal for the multiplexer, OC0 count pulses
for motor0
break;
}
}
void SplitString(void){
bFlagTransmit=true;
char cBuffor[4]={0,0,0,0};
switch(cReceivedBuffer[0]){ //obtain kind of speed regulation
case 'V' : //voltage regulation (open-loop control system)
bFlagPidIsOn=false; //disable PI regulator
for(int n=0;n<5;n++) cBuffor[n]=cReceivedBuffer[n+3];
switch(cReceivedBuffer[1]){ //obtain motor number
case '0': //motor 0
switch(cReceivedBuffer[2]){ //obtain motor direction
case 'R': //right
PORTD &= ~(1<<PORTD2); //change direction
OCR1A=atoi(cBuffor); //obtain value of supply voltage
break;
case 'L': //left
PORTD |= (1<<PORTD2); //change direction
OCR1A=atoi(cBuffor); //obtain value of supply voltage
break;
}
break;
case '1': //motor 1
switch(cReceivedBuffer[2]){ //obtain motor direction
case 'R': //right
PORTD &= ~(1<<PORTD3); //change direction
OCR1B=atoi(cBuffor); //obtain value of supply voltage
break;
```

```
case 'L': //left
PORTD |= (1<<PORTD3); //change direction
OCR1B=atoi(cBuffor); //obtain value of supply voltage
break;
}
break;
}
break;
case 'S' : //speed regulation (close-loop control system)
for(int n=0;n<5;n++) cBuffor[n]=cReceivedBuffer[n+3];
switch(cReceivedBuffer[1]){ //obtain motor number
case '0': //motor 0
switch(cReceivedBuffer[2]){ //obtain motor direction
case 'R':
PORTD &= ~(1<<PORTD2); //change direction
iPresetRPM[0]=atoi(cBuffor); //obtain value of preset speed
break;
case 'L':
PORTD |= (1<<PORTD2); //change direction
iPresetRPM[0]=atoi(cBuffor); //obtain value of preset speed
break;
}
break;
case '1': //motor 1
switch(cReceivedBuffer[2]){ //obtain motor direction
case 'R':
PORTD &= ~(1<<PORTD3);//change direction
iPresetRPM[1]=atoi(cBuffor); //obtain value of preset speed
break;
case 'L':
PORTD |= (1<<PORTD3);//change direction
iPresetRPM[1]=atoi(cBuffor); //obtain value of preset speed
break;
}
break;
}
bFlagPidIsOn=true;//enable PI regulator
break;
case 'C' : //enable/disable communication
if((atoi(cReceivedBuffer))==1) (
```

```
bFlagTransmit=true;
iFramesRecived=0;
iFramesError=0;
iFramesTransmitted=0;
}
if((atoi(cReceivedBuffer))==0){
bFlagTransmit=false;
}
break;
case 'P' : //obtain value of proportional gain
fKp=atoi(cReceivedBuffer)/(float)100;
break;
case 'I' : //obtain value of integration gain
fKi=atoi(cReceivedBuffer)/(float)100000;
break;
case 'T' : //obtain value of the integration time interval
cNumberOfIterations=(char)atoi(cReceivedBuffer);
break;
case 'G' : //send statistical informations
bFlagSendInfo=true;
break;
case 'F' : //ramka odpowiedziala za odebranie informacji o częstotliwośći nadawania
cBuffor[0]=cReceivedBuffer[1];
cBuffor[1]=cReceivedBuffer[2];
cTransmitionDelay=(char)atoi(cBuffor);
bFlagTransmit=true;
break;
}
cReadChars=0;
for(int n=0;n<sizeof(cReceivedBuffer);n++) cReceivedBuffer[n]=0;//reset buffor
}
void SendSpeedVoltage(int iSendNumber){
char cStringToSend[13];
iSaturation[0]=(int)100*((OCR1A)/(float)(1023));//obliczanie aktualnego wypełnienia
PWM dla silnika0
iSaturation[1]=(int)100*((OCR1B)/(float)(1023));//obliczanie aktualnego wypełnienia
PWM dla silnika1
sprintf(cStringToSend,"SV%d%4.d,%3.d",iSendNumber,iActualRPM[iSendNumber],iSaturati
on[iSendNumber]);
sprintf(cStringToSend,"%s,%d\r\n",cStringToSend,USART_checkSum(cStringToSend));
iFramesTransmitted+=USART_sendString(cStringToSend);
}
void SendInformation(void){
//preparing statistical information to send to supervisor application
char cLine[36];
sprintf(cLine,"I%6.lu",iFramesRecived);
sprintf(cLine,"%s,%6.lu",cLine,iFramesTransmitted);
sprintf(cLine,"%s,%4.d",cLine,iFramesError);
sprintf(cLine,"%s,%6.u",cLine,(int)(iWorkTime/(float)cTimeInterval));
sprintf(cLine,"%s,%6.d\r\n",cLine,USART_checkSum(cLine));
iFramesTransmitted+=USART_sendString(cLine);
```

```
}
int main (void){
//PINS ROLE AND CONFIGURATION:
//OC0- PORTB-0-pulses counter from encoders
//RX- PORTD-0
//TX- PORTD-1
//DIR0 PORTD-2-switching the demultiplexer, switching PWM signals for L293D,
changing the direction of motor 0 rotation
//DIR1 PORTD-3-switching the demultiplexer, switching PWM signals for L293D,
changing the direction of motor 1 rotation
//OC1A- PORTD-4-PWM channel A, controlling signal for the motor 0
//OC1B- PORTD-5-PWM channel A, controlling signal for the motor 1
// PORTD-6
//EncoderPORTD-7-switching the multiplexer, switching input signals for the Timer 0
input (OC0)
//ports configuration
DDRD = 0xFF;
PORTD = 0xFF;
DDRB = 0xFE;
int iSendMotorNumber=0;
cScalingFactor=(char)(60*cTimeInterval/(float)cEncoderResolution);
iPresetRPM[0]=iPresetRPM[1]=0;
char cStringToSend[13];
char cInformationDelay=0;
USART_init(19200,8); //initialize serial communication
Timer0_init(); //counting pulses
Timer1_init(8); //generating the PWM signals
Timer2_init(); //generating constant time periods
sei();
while(1){//------------------------------------------------------------------INFINITE
LOOP
if(bFlagRegulation==true){
Regulation(); //enable PI regulator
bFlagRegulation=false;
}
if(bFlagToSend==true){
//preparing information about motors speed and supply voltage
iSaturation[0]=(int)100*((OCR1A)/(float)(1023)); //computing the actual percent
value of duty cycle PWM signal for the motor 0
iSaturation[1]=(int)100*((OCR1B)/(float)(1023)); //computing the actual percent
value of duty cycle PWM signal for the motor 1
sprintf(cStringToSend,"SV%d%4.d,%3.d",(int)iSendMotorNumber,iActualRPM[iSendMotorNu
mber],iSaturation[iSendMotorNumber]);
sprintf(cStringToSend,"%s,%d\r\n",cStringToSend,USART_checkSum(cStringToSend));
iFramesTransmitted+=USART_sendString(cStringToSend); //send infomation about
speed&volate
bFlagToSend=false;
cInformationDelay++;
if(cInformationDelay==20) bFlagSendInfo=true; //send statistical informations
if(iSendMotorNumber==0) iSendMotorNumber=1; //switch motor number
```