A GENERIC REAL-TIME PROCESS CONTROL SYSTEM

by

EDWIN KEITH ARRANT, B.S. in P.E., B.S. in E.E.

A THESIS

IN

ELECTRICAL ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

Approved

December 1989

ACKNOWLEDGMENTS

I would like to thank Dr. Micheal Parten, Dr. Bill Marcy, and Dr. Donald Gustafson for their time and help during my graduate work at Texas Tech. Without their assistance, this project would not have been possible. In addition, I would like to thank FSI International and the technical staff at the Advanced Technology Center for providing the opportunity to implement the control system software on their "Excalibur" process equipment.

CONTENTS

ACKNOWLEDGMENTS		ii
ABSTRACT		iv
LIST OF TABLES		v
LIST OF FIGURES		vi
I.	INTRODUCTION	1
II.	FACTORY AUTOMATION	9
III.	GENERIC CONTROL SYSTEM	22
IV.	GENERIC PROGRAM STRUCTURE	40
V.	DESIGNING AN APPLICATION	77
VI.	GENERIC IMPLEMENTATION	83
VII.	CONCLUSIONS AND RECOMMENDATIONS	90
LIST OF REFERENCES		98
APPENDIX		100

ABSTRACT

A major problem facing manufacturers is the design and implementation of flexible automation systems. This problem is complicated by the many differing requirements for automated systems. Every facility has its own specific requirements; therefore, a generic factory control system design could provide the flexibility and adaptability to solve a wide variety of automation needs. Since factory automation systems are implemented from the bottom up, it is necessary to ensure that the initial automated subsystems are going to be compatible with future system enhancements.

This thesis describes the various levels of factory automation and establishes a functional specification for each subsystem in the automated facility. These specifications provide the general information required to define the design objectives for the facility subsystems. The first step toward obtaining a solution to the automation problem is to develop a generic software control system for process equipment. A first-generation control system has been developed and applied to a four-module HF vapor etcher system, used in the processing of semiconductor wafers.

LIST OF TABLES

4.1	Queue Table	52
4.2	Process Table	53
4.3	Digital Sequence Table	54
4.4	Analog Sequence Table	55
4.5	Digital I/O Hardware Address Table	56
4.6	Analog Hardware Address Table	60
4.7	Safety Table	73

LIST OF FIGURES

2.1	Typical Work Cell Block Diagram	12
2.2	Automated Facility Block Diagram	14
3.1	Typical Finite State or Mealy Machine	24
3.2	State Diagram for Simple Sequential Machine	26
3.3	State Diagram of Control Program	28
4.1	Multitasking Software Structure	41
4.2	Flow Chart of Main Program	46
4.3	Example I/O Operations	59
4.4	Flow Chart of "Sequence Manager"	62
4.5	Flow Chart of "Execute Finite State Machine"	65
4.6	Flow Chart of "Digital Input Verify"	67
4.7	Flow Chart of "Safety Manager"	74

CHAPTER I

INTRODUCTION

Automation has become the method by which manufacturers can most effectively reduce cost and increase production, in order to remain competitive [1]. In addition, automation also is needed to quickly install new technologies for manufacturing advanced products. Further research in the area of automation is required to provide the level of control needed by manufacturers. Existing automation systems have proven to be lacking in functionality, expensive to purchase and implement, difficult to maintain, and insufficiently flexible to adapt to changing requirements. In order to achieve the required level of automation, new systems have to be designed and implemented [2].

A hierarchical approach to automation is suggested by many experts in the field of Computer Integrated Manufacturing (CIM). In order to develop a framework from which to design and build automated systems and subsystems, a detailed "top-down" factory design that identifies all functional and informational flow requirements from the operational and management perspective must be completed. This plan must establish certain design requirements for all the automated equipment that is to become a part of the overall system. Once a general plan has been established,

implementation would occur from the equipment level toward the factory level. This "bottom-up" implementation procedure should follow the design constraints defined by the top-down plan, so that each individual piece of process equipment may be adaptable into the complete automated system [3].

Problem Description

A major problem facing manufacturers is the design and implementation of a flexible automation system that will reduce production cost, improve quality, and reduce delivery time [2]. This problem is not simple and requires much analysis, insight, and cooperation from manufacturers and equipment vendors. Experts in computer systems, materials tracking and handling, scheduling, process control, communications, and data logging must provide input into this complex design problem [3]. The problem is complicated by the many differing requirements for automated systems. Every facility has its own specific requirements. A generic factory control system design would provide the flexibility and adaptability to solve a wide variety of automation needs. Since factory automation systems are implemented from the bottom up, it is necessary to ensure that the initial automated subsystems will be compatible with future system enhancements [4].

Top-Down Design Approach

The design of the factory automation subsystems can begin once the functional requirements are established. The first phase of the problem solution is to develop a fully automatic, generic control system for the process equipment according to the top-down plan. The equipment, in order to be flexible for both present and future applications, should be capable of being controlled in a stand alone mode by an operator or controlled remotely by a master controller. This level of automation would provide the required automatic process control while also providing for future expansion of the factory control system.

The second phase of the overall plan for solving the factory automation problem is to establish "islands of automation" or work cells containing related process equipment. Each work cell would be controlled by a work cell controller (WCC) computer that manages the automatic process equipment within the work cell. An example from the semiconductor industry would be a photolithography cell, which might contain coaters, developers, and steppers. Other work cells could be formed from process equipment in the areas of diffusion, chemical vapor deposition, ion implantation, and metalization [1].

Each work cell would be monitored and controlled by a Work Cell Controller (WCC) computer which would be responsible for coordinating all the activity in the work cell.

The WCC must provide the most efficient use of the equipment in the work cell, while at the same time reducing the amount of time required for the material to be processed through the work cell. The WCC must provide the following capabilities [5,6]:

1. Communication with the factory host and process equipment in the work cell.

2. Monitor and control the loading and unloading of material into and out of the work cell.

3. Schedule the movement of material from one process equipment to another.

4. Manage the material transfer mechanism within the work cell.

5. Monitor work cell and process equipment status.

6. Monitor material location and status.

7. Accumulate historical performance data.

Several work cells could be monitored and controlled by a factory host to form a fully automated manufacturing facility. The third phase of the top-down plan would establish a factory control system containing various work cells. The factory host is responsible for managing each WCC in a similar way that the WCC computer manages each automatic process equipment in the work cell.

Previous Work

There has been a dramatic increase in the number of independent computer systems for process control due to the continuing decline in the cost of computing [7]. The popular and relatively inexpensive microcomputer is moving into industrial control applications at an increasing rate. Reasons for this increase are the improved price to performance ratio and the availability of the required data acquisition and control interface hardware. Communications standards are also being developed, which are providing the needed tools for linking the process equipment to the WCC and the WCC to the factory host and other systems.

Many equipment vendors provide automatic controlled process equipment, but often without following a high level or top-down plan [3]. Most control system software designs are a "one-of-a-kind" design, which will work for only one system. The problem with such systems is that these systems are not flexible nor easily adaptable for future applications. The introduction of new equipment requires that new software be developed, resulting in time delays and added costs. A generic process control software system is needed that will provide a high degree of flexibility and adaptability for most process control applications.

Objectives

As stated previously, the solution to the factory automation problem is very complex and will require the cooperation of many experts [3]. A top-down factory automation design which identifies the functional specifications of each section of the control system is presented in Chapter Two. This design is presented to clearly establish facility subsystems requirements so that the specific design objectives of these subsystems can be defined.

The main objective of this thesis is to present the design of a generic process equipment control system. This equipment control system provides a partial solution to the flexible automation design problem. A portion of this design has actually been implemented and the equipment is presently operating in several semiconductor manufacturing facilities worldwide.

This flexible equipment control system provides the capability to reconfigure a process sequence without having to revise the control program source code. In order to provide this flexibility, the specific equipment and process dependent variables such as, input and output hardware addresses, process sequences, safety sequences, maintenance sequences, operator or WCC interfaces, and communication interface specifics are arranged in programmable data tables. It is these data tables that drive the source code to control the process equipment and provide interfaces to

other systems. A data table editor routine is used to customize the control software for the particular application. The equipment designer can customize the process control program by completing the data tables with equipment and process specific information. Equipment or process revisions do not require source code revisions. Control program revisions are made by simply revising the data tables which drive the generic equipment control program [8].

Thesis Contents

The following chapter provides an overview of the various levels of factory automation and establishes the functional specification for each subsystem in the automated facility. These specifications should provide the general information required to define the objectives for the design of the facility subsystems, specifically, the generic equipment control system. The specifics of the generic equipment control system and the various system controlling parameters are discussed in Chapter Three.

The software data structures play a very important role in this generic equipment control system. In order to provide flexibility for many process control applications, the data structures must be well planned. The program source code is designed to operate on the information programmed into the data tables. Since the program source

code itself is designed to be generic, all process and hardware functionality must be programmable via the data structures. The software configuration and data structures are discussed in Chapter Four.

The procedure for the design of an equipment control system, using the generic program, is discussed in Chapter Five. This discussion covers the method of designing the overall control of the equipment and the method used to implement the control with the generic program.

An actual equipment control application was programmed using the generic control system. This results of this test implementation is covered in Chapter Six.

Chapter Seven presents the thesis conclusions and recommendations for future software improvements. These recommendations resulted from the experience gained from the actual implementation of the generic equipment control system. This final chapter sums up the results of this design effort and makes final remarks concerning the effectiveness of generic factory and process control.

CHAPTER II

FACTORY AUTOMATION

To effectively increase production and reduce cost, manufacturers are looking to ever increasing levels of automation [1]. Manufacturers must decide on the most effective use of automation in order to remain competitive.

Levels of Automation

There are various levels of automation, some of which are readily available today. These levels may be divided into four groups. The first and lowest level consist of semiautomatic process equipment which is controlled by one or several programmable controllers. Level two contains fully automatic equipment. Integrated work cells make up level three and level four consists of several work cells that are integrated into a fully automated facility. Each level of automation will be described in detail in the following sections.

Semiautomatic Equipment

Semiautomatic process equipment makes up level one. Equipment of this type is controlled by one or more programmable controllers. The operator may be required to make manual settings for process time, temperature, pressure, or flow rate. The programmable controller sequences

the machine through its various steps after the operator has manually setup the equipment. Equipment of this type may only be run by an operator since it has no remote control capabilities. Without remote control capabilities, process equipment cannot become a part of an integrated work cell; therefore, equipment of this type is not compatible with the top-down factory automation design.

Fully Automatic Equipment

The second level includes fully automatic process equipment. This type of equipment requires little operator intervention. The operator simply loads the material into the machine and selects an appropriate process or "recipe." A recipe is a specific processing program that is used by the equipment to prepare the material for the next processing station. Once the material has completed the process, the operator unloads the material and transfers it to the next processing station. The operator is not required to make any adjustments or manual settings to the equipment. Some fully automatic equipment designs provide closed loop control of certain process variables. This capability allows the equipment to maintain process variables within desired limits. The controller will shut the system down if these variables cannot be maintained within specified limits.

Fully automated equipment provides several time and cost saving features. Process consistency and repeatability are much improved when the operator is not required to set important machine or process parameters. Safety is also improved in many situations, by isolating the operator from potentially dangerous conditions. Fully automated systems provide a significant safety and cost advantage over the semiautomatic systems.

In order to provide compatibility within a work cell environment, these fully automatic systems would have to provide remote control capabilities. The equipment control system would be required to have the necessary communication hardware and software interfaces to allow for remote-controlled applications. To enhance flexibility, such equipment should be controllable by an operator or WCC.

Work Cell

The integration of a group of related fully automatic process equipment into an "island of automation" or a work cell is the next level of automation. This third level requires a work cell controller (WCC) computer which manages all of the activity within the work cell. The work cell is comprised of a WCC computer, fully automatic and remote-controlled process equipment, a material transfer mechanism, and a communication network. Figure 2.1 shows a block diagram of a typical work cell.

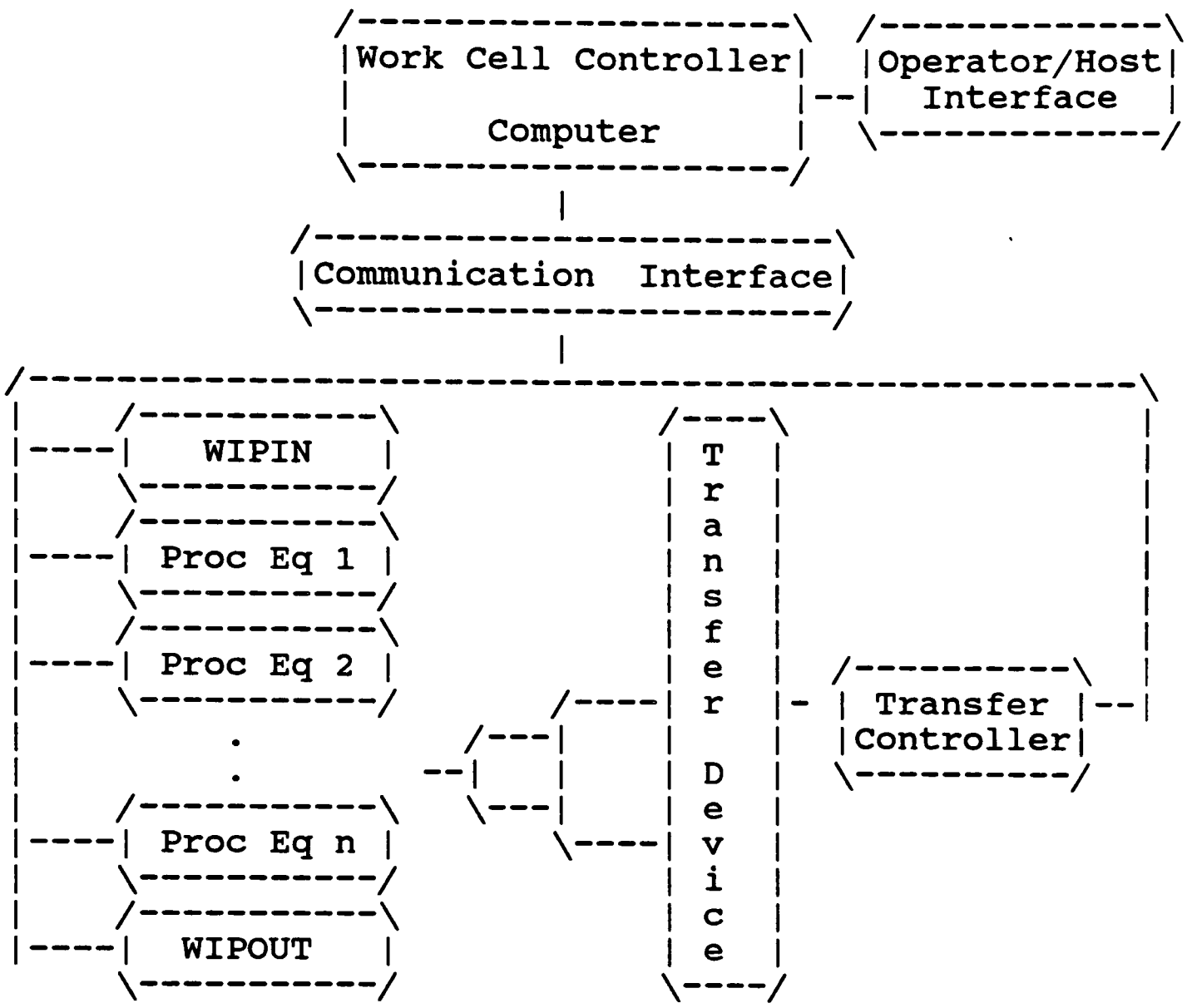


Figure 2.1 Typical Work Cell Block Diagram [4]

Material is loaded, by the operator, into the work in process input area, referred to as the WIPIN. The operator, via the operator interface, is responsible for selecting the proper process station sequence and recipes that are to be run at each process station for the material placed into the WIPIN. The WCC will command the material transfer mechanism to move the material from the WIPIN to each processing station, to complete the process sequence selected by the operator. Once the material has completed all

specified processes, it is transferred to WIPOUT. The work cell WIPOUT is unloaded by the operator.

There are many advantages to integrated islands of automation. Due to the computing power of the WCC, efficient scheduling of the material through the work cell is possible. This feature provides better equipment utilization, reducing the idle time of the equipment and the time required for the material to be processed through the cell. Automatic recipe selection ensures proper processing, reducing the opportunity for operator error. Automatic and accurate data logging is also an added benefit of the work cell system.

Facility Level

The ultimate or highest level of automation is the factory level, where several work cells are combined together to form a completely automatic fabrication facility. The host controls activity in the facility in the same way as the cell controllers control activity in the work cells. The factory host manages the flow of material into and out of the individual work cells. This system would consist of a factor host computer, a communication network, WCC computers and the associated automatic process-controlled equipment, materials transfer mechanisms, plus peripheral devices such as mass storage for data logging and printers for making hard copies of desired information. In the

automated facility, operators have the responsibility of loading the material into the facility WIPIN and unloading it from the WIPOUT after the material has completed all the processes for that facility. In addition, the operator must input the information needed by the system to properly process the material. A block diagram of an fully automatic fabrication facility is shown in Figure 2.2 [4].

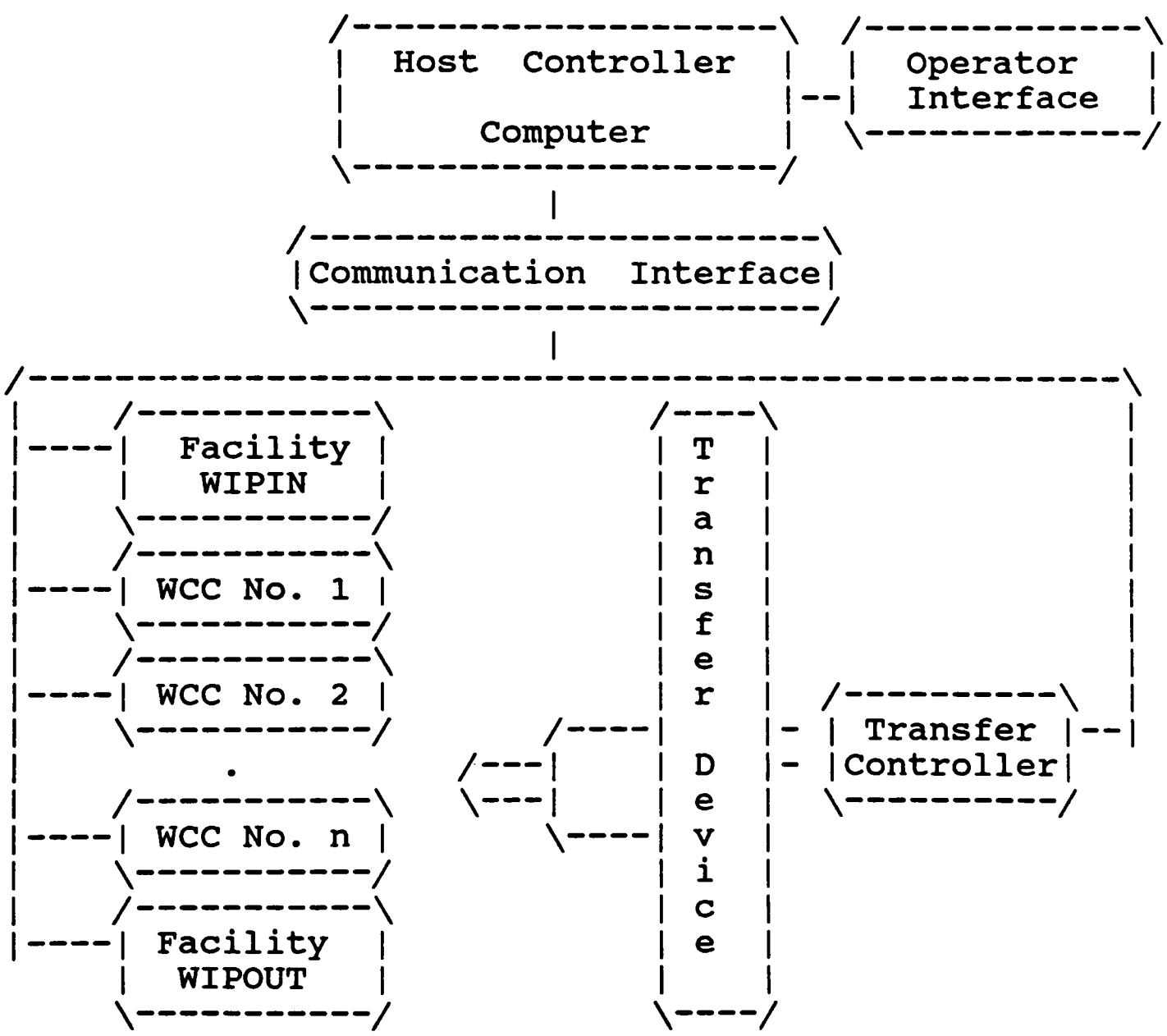


Figure 2.2 Automated Facility Block Diagram [4]

System Functional Requirements

To establish such a complex system, much planning is required. Experts in computer systems, plant management, data base management, materials transfer and tracking, scheduling, process control, and network communications must be consulted in order to accomplish the objective of a fully automated facility [3]. A general, top-down design must be developed which defines the functional requirements of each subsystem. The relationships between each subsystem must also be defined so that the interfaces can be more easily understood. A functional description of the factory subsystem follows.

Host

The factory host is responsible for managing the activity within the entire system. The basic responsibilities of the host are to schedule and control the material transfers from one work cell to the next, maintain a detailed historical data log of all important system parameters, communicate with all the subsystems on the network, and provide a user interface. The scheduling algorithm could apply a combination of rules, such as, first in first out, assigned material priority, process equipment availability, transfer mechanism status, plus other rules, in order to most efficiently route the material from one work cell to the next. The WCC will implement a similar

scheduling algorithm to route material from one process equipment to the next within the work cell. The material transfer mechanism could be a robot or other mechanism which can be controlled remotely over the communication network by the host. Communications between the host and WCC computers, robots, and network peripherals would be required. Historical data logging must also be provided in order to keep track of material location and status, equipment and process status, and fault records. An operator interface is required to allow the user to obtain whatever status information is needed, to start material processes, to revise recipes, and to revise system process parameters.

Network Communication Interface

The communication network provides the necessary two-way link between the subsystems in the facility. This network would be required to provide reliable communications in hostile and noisy environments, support several hundred nodes, operate well under high data traffic conditions, and allow for easy growth and reconfiguration [1,9].

A well defined communication protocol is required so that subsystem designers can incorporate network capabilities into these systems. Such a protocol must establish certain communication rules and message formats. A network operating system would set up the entire communication system, defining each node as a WCC, process equipment,

transfer mechanism, host, or network peripheral. The network operating system would support recipe and file transfer, in addition to command and status communications.

Some specific communication networks available today are Manufacturers Automation Protocol (MAP) [10] and Technical and Office Protocol (TOP). These two are high baud rate networks and have the capability to handle the high data throughputs required in complex factory systems. Both MAP and TOP are based on the seven-layer Open Systems Interconnect (OSI) model for computer communications. The Semiconductor Equipment and Material Institute (SEMI) is implementing Semiconductor Equipment Communication Standard (SECS I/II) protocol, which uses RS-232 as the hardware layer [2,11]. This standard provides some needed functionality, but is limited due to low data transfer rates.

Material Transfer Mechanism

The host must have a means for moving material from one work cell to another; therefore, some type of material transfer mechanism must be employed. This mechanism could be a robot or other machine that can be controlled by the host over the network. The transfer device would be required to supply detailed status information when queried by the host so that the host could make the most efficient use of the device. Communication protocol would have to be defined in such a way that all necessary handshaking could

take place between the host and the transfer device controller.

Work Cell and Work Cell Controller

The work cell controller manages all the activity within the work cell based on the directions received from the facility host. As previously mentioned, the WCC schedules and controls the material transfers, maintains cell historical data logs, and communicates with the operator or host and with the equipment within the cell. In the fully automated facility, the host operates the WCC instead of the operator. The functionality of the work cell is the same in both cases, except for the substitution of a host interface in place of the operator interface. A flexible WCC system design would allow for both a host interface and an operator interface to allow for work cell operation in either case.

Considerations

A flexible factory automation system requires that the system remain operational in the event of inevitable subsystem failures. The facility must be able to continue to process material independent of the factory host, one or more material transfer mechanisms, one or more WCCs, and the communication network. If a subsystem is down, the throughput will certainly be reduced, but the processing of the material through the facility should not be suspended.

This criterion requires much flexibility to be incorporated into the facility system design.

The fully-automatic equipment must provide two modes of operation, an operator controlled mode and a remote-controlled mode. If all facility process equipment can be controlled by an operator or remotely by a WCC, much flexibility is available for future expansion. Since automation cannot take place all at once and since implementation logically must begin at the equipment level; it is clear that a flexible and generic equipment control system is needed. Such a system would supply the process control requirements, while allowing for future expansion to higher levels of automatic control.

First Task

To obtain a solution to the factory automation problem, it is first necessary to replace the stand alone process equipment with equipment that can become a part of the automated facility. This task requires that the machine be designed so that it can be loaded by an operator or a robot. The equipment must be fully automatic and be controllable by an operator or remotely by a cell controller. In addition to providing the functionality to satisfy the process requirements, the equipment control system should be designed to provide flexibility and adaptability for a variety of equipment control needs.

Most equipment control systems perform similar functions. These systems have digital and analog inputs and outputs which must be read and set. Safety interlocks must be scanned and safety shut down sequences must be executed in the event an unsafe condition is detected. Process and maintenance sequences must be executed. The process equipment must display status information and accept information and commands from the host or operator. Since most process control software is similar when considering the basic requirements, it should be feasible to design a generic software structure that would work for most applications.

A generic structure would have to provide a way to program the machine and process specific functions. Process equipment specific variables such as input and output hardware addresses, process sequences, safety sequences, maintenance sequences, operator and WCC interfaces, plus other communication interfaces must be arranged in programmable data tables. It is these data tables that contain the specific machine and process data that would drive the generic structured source code to control the equipment. The equipment design engineer would program the equipment by completing the data tables with specific machine and process information. Equipment or process revisions could be made by revising the driving data tables, without having to revise the source program; thereby, saving software

revision time. A generic software structure for process equipment control will solve the first part of the factory automation problem. The first phase of the implementation of a flexible automatic factory system is made possible by the completion of this first task [4]. The detailed functional description of this generic control system is described in the following chapter.

CHAPTER III

GENERIC CONTROL SYSTEM

Presently, most equipment control systems are a "one-of-a-kind" design; that is, these systems are designed for one specific process equipment application [3]. Control systems of this type are not easily adaptable to new equipment designs nor do these systems provide much flexibility for future enhancements. To solve this control problem, a generic process equipment control system is needed in order to facilitate the development of flexible and adaptable automated systems.

Equipment Control System Requirements

Most automatic process equipment performs related functions and it is this common functionality that a generic control system can implement. Equipment control systems must execute selectable and programmable process sequences or process recipes. Safety interlocks must be read and safety shut down sequences must be carried out in the event of a malfunction or the occurrence of an unsafe condition. Maintenance sequences must be run to provide a means to setup, calibrate, and repair the equipment. Controlling inputs to these machines consist of digital and analog signals and operator or WCC inputs. Outputs from the

equipment controller consist of digital and analog signals, messages to the operator display screen or WCC, and commands to slave subsystems, such as, smart motor controllers or other smart slave subsystems. Since most process equipment performs similar functions, a generic program structure can be developed to implement these similar functions. The specific functions of process equipment are different, requiring a generic control system to provide a method to configure the control program to suit the specific needs of the equipment. The method used to provide this capability is to place specific equipment and process dependent variables in programmable data tables. The generic source code program would read these data tables to obtain the information necessary to control the process equipment.

The following sections contain a description of the general requirements for a process equipment controller. The first section includes a discussion of the various sequencers used by the equipment to carry out process, safety, maintenance, start up, and shut down sequences. A Sequence Manager is responsible for initiating the correct sequence. The next section deals with the Operator or WCC Interface. This interface is responsible for two-way communication between the equipment and its master, whether the master be a WCC or an operator. The third section of the control system is the Safety Manager, which scans the system inputs, looking for disallowed or unsafe conditions.

The final section of the basic control system is the Communication Manager, which is responsible for the actual two-way communication with the WCC and with any slave subsystems within the process equipment system.

Sequence Manager

Finite State Machine

A programmable finite state machine or sequencer is needed to control the sequential operation of the process equipment. A finite state machine is a processing system that generates an output sequence that is a function of the present system inputs and the past history of the inputs. Finite state machines are cyclic in nature; that is, these machines repeat operations if run for a long enough time. A finite state machine has a finite number of different conditions or states in which the machine can reside [12]. A block diagram of a finite state machine is shown if Figure 3.1.

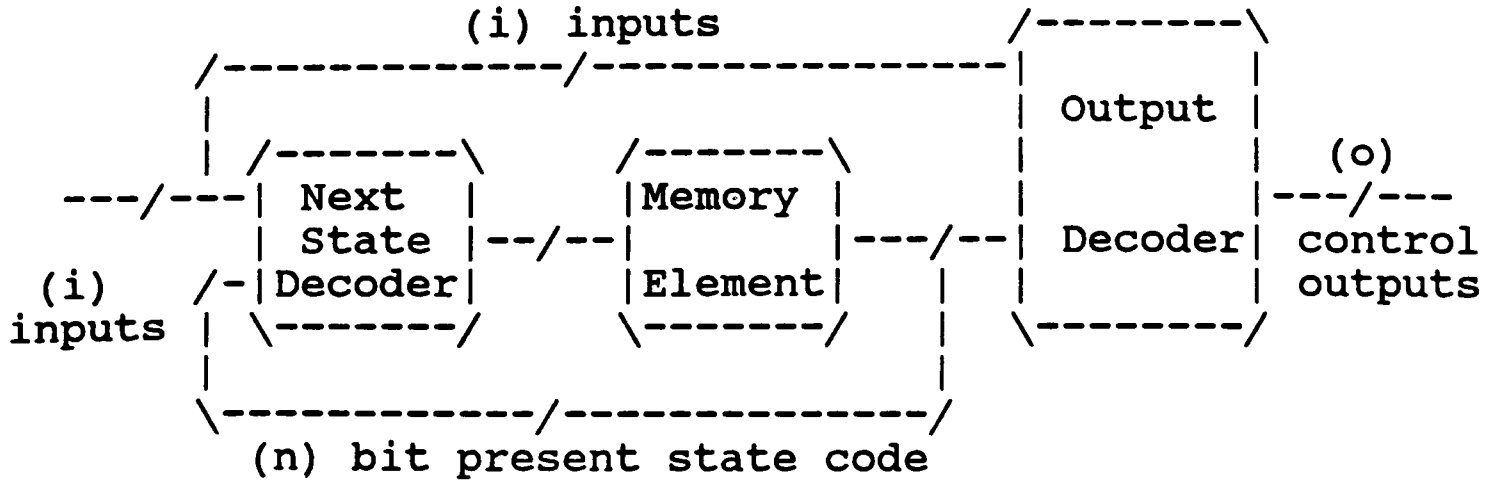


Figure 3.1 Typical Finite State or Mealy Machine [12]

This machine is commonly referred to as a Mealy machine. The sections that make up this machine are the next state decoder, memory element, and output decoder. The (i) inputs to the next state decoder are from the outside world and from the (n) outputs of the memory elements, which represent the present state of the machine. The outputs of the next state decoder are applied to the inputs of the memory block. These outputs can be referred to as the next state code. The next state code will become the present state code after the memory loads and stores it. This process is called a change of state of the finite state machine. The outputs of the memory, or the present state of the machine, along with the inputs from the outside world are applied to the output decoder. The output decoder decodes its inputs and generates (o) controlling outputs, which are sent to the system under control. It can be seen that the next state of a finite state machine is a function of the input history, which determines the present state of the machine, and the current value of the inputs to the machine.

A state diagram is often used to graphically represent the sequential operation of a finite state machine. Bubbles are used to represent the individual states. The number assigned to the state is placed inside of the bubble. State transitions are denoted by arrows pointing from one state to the next state. The input conditions required

to cause a state transition are printed adjacent to the arrow and to the left of the slash (/), while the controlling output values are printed to the right of the slash (/). Figure 3.2 shows a simple sequential machine.

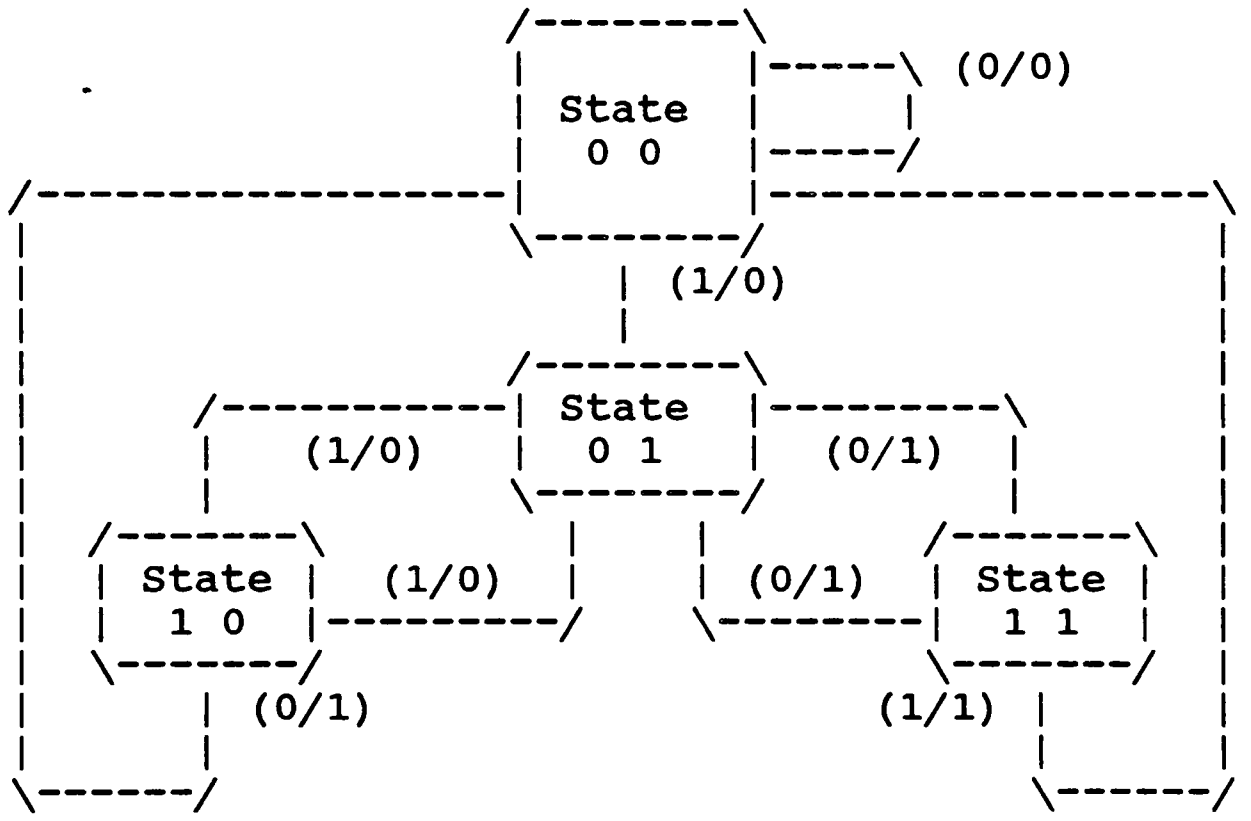


Figure 3.2 State Diagram for Simple Sequential Machine [12]

Assuming the present state of the machine is State 00, the next state of the machine will be State 00 if the input is zero (0). The controlling output will also be zero (0). If the input changes to one (1), a state transition will occur from State 00 to State 01. Once in State 01, the next state will be either State 10 or State 11, depending on the value of the input. By observing the state diagram, it is clear how the sequential operation of the machine

behaves. State diagrams are valuable tools in describing sequential operation of finite state machines.

The programmable sequencer section of the control system can be described as a finite state machine which sequences the equipment from one state to the next state. State changes occur when all the required input conditions for a state transition become true. Input conditions such as digital and analog input values, operator or WCC commands, slave subsystem status, timer expirations, counter values, and flag values can act as state input conditions which are used to determine state transitions. The outputs from the finite state machine can be digital and analog control signals, messages to the operator display screen or to the WCC, commands to equipment slave subsystem controllers, starting of timers, incrementing of counters, and the setting of flags.

From the state diagram in Figure 3.3, it can be seen that state transitions occur when certain input conditions become true. The transition from State N - 1 to State N sets some digital and analog outputs and starts a timer, which will be used to detect a time-out condition in State N. While in State N, two next states are possible, depending upon the input conditions. If the input conditions for a transition to State N + 2 become true before the timer (time-out) expires, the next state will be State N + 2.

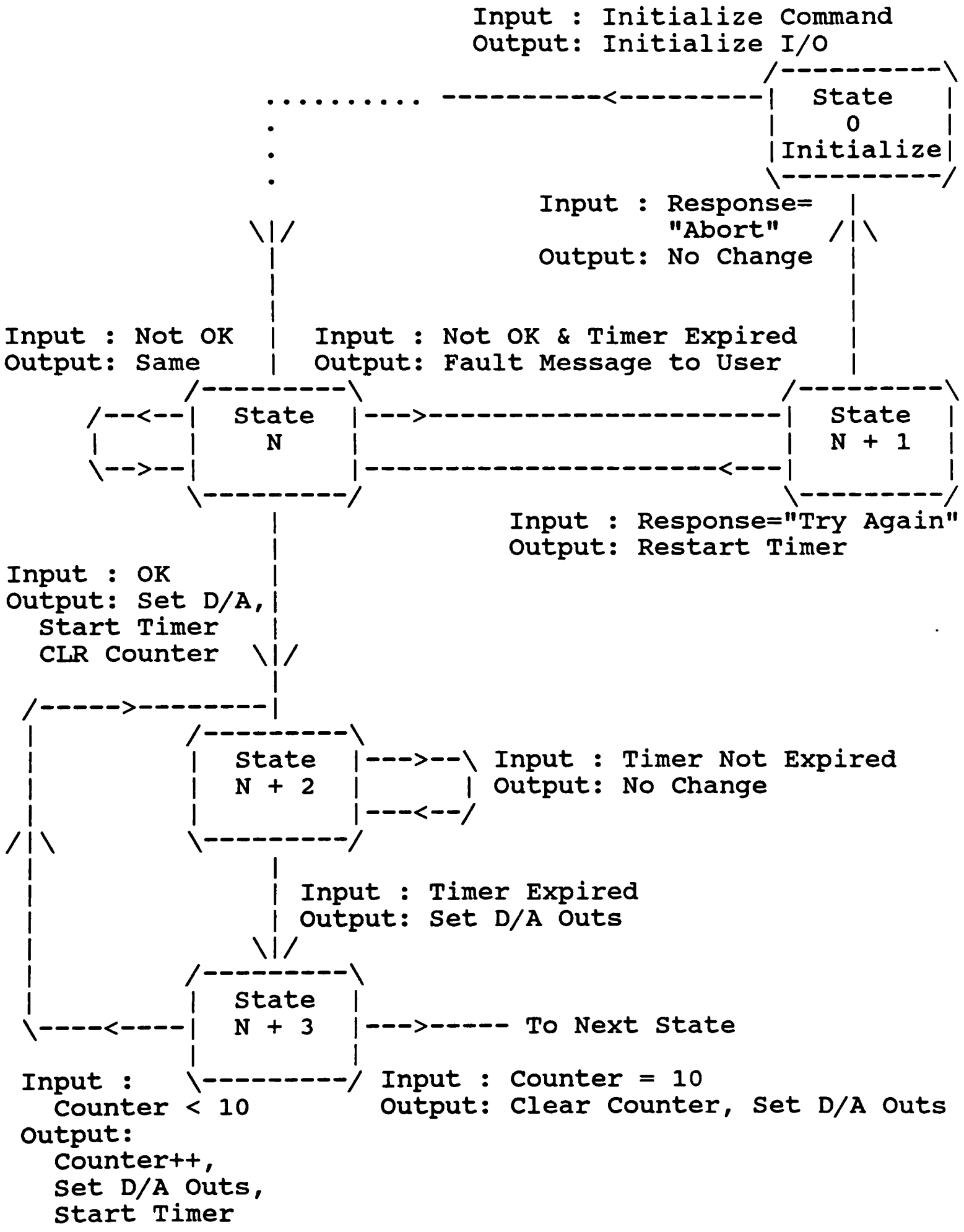


Figure 3.3 State Diagram of Control Program

If the timer expires before the input conditions become true, the next state will be State N + 1. At the transition to State N + 1, a message will be sent to the operator or WCC indicating that a fault has occurred. The finite state machine will remain in State N + 1 until the operator or WCC acknowledges the message with either a "Try Again" or an "Abort." The next state, after State N + 1, depends upon the message received from the operator or WCC.

Assuming that the input conditions are valid so that the machine can sequence from State N to State N + 2, a timer (time delay) is set along with some digital and analog outputs. The machine will remain in State N + 2 as long as the timer has not expired. Once the timer expires, the machine will change states to State N + 3. The machine will remain in State N + 3 for a short time; that is, for "one clock cycle" or one pass through the code. The next state is determined by a counter. If the counter is not equal to 10, the next state is State N + 2. The digital and analog outputs are set and the timer is started. The counter will cause State N + 2 and State N + 3 to be executed 10 times. Once the counter has reached 10, it is cleared and the machine sequences to State N + 4.

The programmable finite state machine provides the control necessary to perform all the sequential operations of the equipment. To provide for programmability, the input conditions required for state transitions along with

the output conditions associated with the state transitions must be stored in programmable data tables. A data table editor program is required to view and edit these input and output conditions. The data table editor is not a part of the control system software, but only serves as a tool to customize the system for the particular application.

Required Sequencers

There are five classifications of sequential operations which are needed by most equipment control systems. These operations can be divided into the following five groups.

Process sequencer. The primary function of the equipment is to carry out a process sequence or a series of operations leading to some desired result. The equipment operates on the material according to a specific and user-selected process, referred to as a "recipe." The recipe supplies all the specific information needed by the Process Sequencer to prepare the material for the next processing stage.

Safety sequencer. A safety sequence is executed in the event of a detected fault or malfunction, in order to put the machine in a safe state. Safety sequences are automatically initiated by the Sequence Manager when an unsafe condition occurs for the purpose of protecting the

facility personnel, the equipment, and the material being processed.

Maintenance sequencer. To maintain the equipment in proper operating condition and calibration, it is periodically necessary to run specific maintenance sequences. These sequences are often run at regular intervals to calibrate the equipment. In some cases, maintenance sequences are used to prepare the machine for maintenance or to assist the maintenance personnel in performing maintenance operations.

Start up sequencer. When a machine is brought up for the first time or from a "cold-start," a start-up or initialization sequence may be required. Such an operation maybe needed to prepare the machine for normal processing.

Shut down sequencer. Before a machine is taken off line, a shut down sequence may be run in order to put the machine in the proper state for transportation or non-use.

Sequence Manager

A programmable finite state machine or sequencer can provide the needed functionality to control all the process, safety, maintenance, start up, and shut down sequences for most process equipment. A Sequence Manager is required to monitor and control the programmable finite state machine so that the proper sequences can be executed at the desired time. In some cases, the Sequence Manager

is required to suspend the execution of a sequence. If a fault occurred that prevented a sequence from completing, the Sequence Manager might simply suspend operation and notify the operator or WCC that a fault had occurred. Once the fault has been corrected, the Sequence Manager could restart the suspended sequence at the state where the fault occurred. In other cases, the occurrence of a fault may require that the current sequence be suspended and a safety sequence be initiated. In such a situation, the suspended sequence would not be restarted at the state where the fault occurred. Sequences that are interrupted would be restarted from the beginning, once the safety sequence terminates and conditions are normal. Operator or WCC commands can also interrupt a sequence. The Sequence Manager would also be responsible for monitoring commands from the Operator/WCC Interface and determining the proper control to be applied to the programmable finite state machine.

To expand the flexibility of the sequencer, special timers and counters could be used by the Sequence Manager to determine which sequence to initiate. For example, some equipment must be periodically calibrated. In some cases these calibration routines may be run automatically, when the equipment is not processing material. To provide the necessary stimulus to the Sequence Manager, the expiration of a timer may be used to initiated the calibration routine at an available point in time. Counters may also be used

in a similar manner. For example, if a maintenance sequence is required after a certain number of process cycles, a counter can indicate to the Sequence Manager that the maintenance routine is required. Counters may also be used to control looping operations. The initiation of periodic calibration or maintenance routines may need to be acknowledged by the operator or WCC prior to execution. In these situations, the Sequence Manager would request permission, via the Communication Manager, before the initiation of the routine. Once permission was received, the routine is executed.

Operator/WCC Interface Manager

The Operator/WCC Interface allows the machine to be accessible either directly by an operator or indirectly by a WCC. In both cases, this interface performs basically the same function, which is to provide a link between the equipment and its master. Complete equipment and material status information should be provided by the equipment to either the operator or the WCC. The WCC will use this status information to more efficiently schedule the flow of material through the work cell. In addition, the WCC will keep a historical data log of important equipment status to provide valuable information for later analysis. The operator must also have access to the machine status to aid in scheduling of his or her work.

The equipment should also be able to accept commands from the operator or WCC so that material may be processed and the desired equipment and material status information can be obtained. The WCC will command and query the process equipment via the Communication Manager by sending commands and messages. The operator will input data and commands by way of a keyboard or similar interface. The equipment will respond to an operator by displaying information to a computer screen or similar display.

The Interface Manager should provide detailed information about the status of the equipment, the process, and the material being processed. The equipment status report should include the state of each input and output in the system plus any faults that may have occurred. The present state of the machine should also be available. The status of the process should also be available upon request. This status report should relate the present state of the Process Sequencer, the remaining time in that state, and the time remaining until the material is to complete processing. The material status report contains information concerning the material being processed; such as, the amount of material that has completed processing, the process recipe used to process the material, and other data pertaining to the processing of the material. Equipment faults should also be reported with a time/date stamp.

In keeping with the overall design of a generic control system, it is also necessary to design a table-driven Operator/WCC Interface. This programmable Operator/WCC Interface would require a supporting off-line screen configuration routine that would define the arrangement of the process, maintenance, and other information screens. This routine would not be a part of the process control program, but would only serve as a tool to configure the data tables that drive the Operator/WCC Interface.

Normally, information displayed on the screen can be classified as static or dynamic information. Static information can be labels, titles, boxes, or graphics that do not change with state changes of the machine. Dynamic information consists of displayed data that does change with time or as the machine cycles through its various states. The various screens defined by the screen configuration program would be managed and displayed by a screen manager.

In addition to the screen configuration program, table driven operator and WCC interface routines would have to be developed. Message look up tables would have to be designed so that the Sequence Manager could send the proper messages to the operator screens or WCC. Data tables containing valid responses to each message would also have to be designed to provide for flexible user responses to machine prompts.

For operator interfaces, it is necessary to prevent certain users, that do not have the correct authorization, from obtaining access to certain machine functions. This feature prevents operators from accessing maintenance or process screens that are protected. A pass code editor and manager would be required to provide this needed functionality.

The main purpose of the Operator/WCC Interface is to provide the link between the equipment and its master. All required information and commands must flow through this Interface Manager. A generic Interface Manager and its support tools are needed to accomplish the goal of a totally generic control system.

Safety Manager

The responsibility of the Safety Manager is to prevent any unsafe conditions from occurring in the equipment system and to shut the system down in the event that some fault or malfunction causes an unsafe condition. An unsafe condition is defined as any condition that poses a threat to facility personnel, the material being processed, or to the equipment itself. In order to prevent unsafe conditions, the Safety Manager must not allow the operator, maintenance person, or WCC to command the equipment to carry out an operation that would endanger the facility personnel, material, or equipment. In addition, the Safety

Manager must continuously check the equipment safety interlocks. If any of the safety interlocks are triggered, the active sequence must be suspended and an appropriate safety routine must be executed. The Safety Manager must communicate with the Sequence Manager in cases when safety sequences are to be initiated.

To provide for a generic and flexible system, the Safety Manager must also be driven by programmable data tables. The Safety Manager would scan the system inputs, which are stored in the safety table, looking for fault conditions. If any faults were found, the Safety Manager would take the action specified in the table for the specific fault detected. In addition to fixed input conditions that are always disallowed, there may be conditions that are conditionally disallowed. The Sequencer may write disallowed conditions to a dynamic portion of the safety table when these conditions are to be considered as safety concerns by the Safety Manager. Likewise, the Sequence Manager would clear these entries to the dynamic safety table when the conditions were no longer a safety concern.

The objective of the Safety Manager, in all cases, is to protect the facility personnel, material being processed, and the process equipment. The Safety Manager must be executed as often as necessary to ensure adequate response to detected faults.

Communication Manager

The Communication Manager is responsible for providing a way for the various managers that make up the control software to communicate with the WCC and the smart slave controllers in the equipment system. Various communication protocols may be required to provide the flexibility needed by different manufacturers. Many equipment slave controllers use the RS-232 standard for communication. If the equipment control system is to communicate to these slave controllers, RS-232 software drivers are needed. The link between the equipment and the master computer or WCC may be one of several established communication networks, such as MAP or TOP, which are used by some manufacturers in network systems [10]. The semiconductor industry uses SECS-I/II as a network communication system. The task of the Communication Manager is to transfer the required information to and from the process equipment over the required network systems. The Communication Manager is required to format and transmit the information over the network and to decode the received information and route it to the correct destination within the equipment control software. The Communication Manager is responsible for the handshaking between the equipment and outside world systems and between the equipment and its various other software managers [13].

Generic Control System Summary

The preceding explanation defines the basic functional requirements for a generic equipment control system. Since most equipment control systems perform similar functions, it is possible to implement a generic system according to the plan described. This implementation requires the development of data structures that contain the information needed to drive a generic control program and the generic control program itself. In addition, support programs are required to edit the programmable data tables which drive the control program. These generic structures and the control program are discussed in the following chapter.

CHAPTER IV

GENERIC PROGRAM STRUCTURE

An equipment control system is needed that can provide the functional requirements of fully automatic process equipment previously described. This control system must be required to set up and control all equipment functions so that the operator or WCC would only be required to load the material and input the desired process recipe. Such a system design requires only that the functional requirements be satisfied; however, a generic control system design which provides these functional requirements would be extremely valuable if it could be adapted to various process control applications. This chapter describes the overall software structure and the generic or table-driven control program.

Multitasking Control Software Structure

Software design for process control applications is quite involved and requires the development of long and complex programs. In order to reduce software development time and simplify the program design, a multitasking structure is used to aid in the segregation of control system tasks. These tasks are partitioned into modules that perform individual functions [14]. A block diagram of the

tasks. These tasks are partitioned into modules that perform individual functions [14]. A block diagram of the control program running under a multitasking operating system is shown in Figure 4.1.

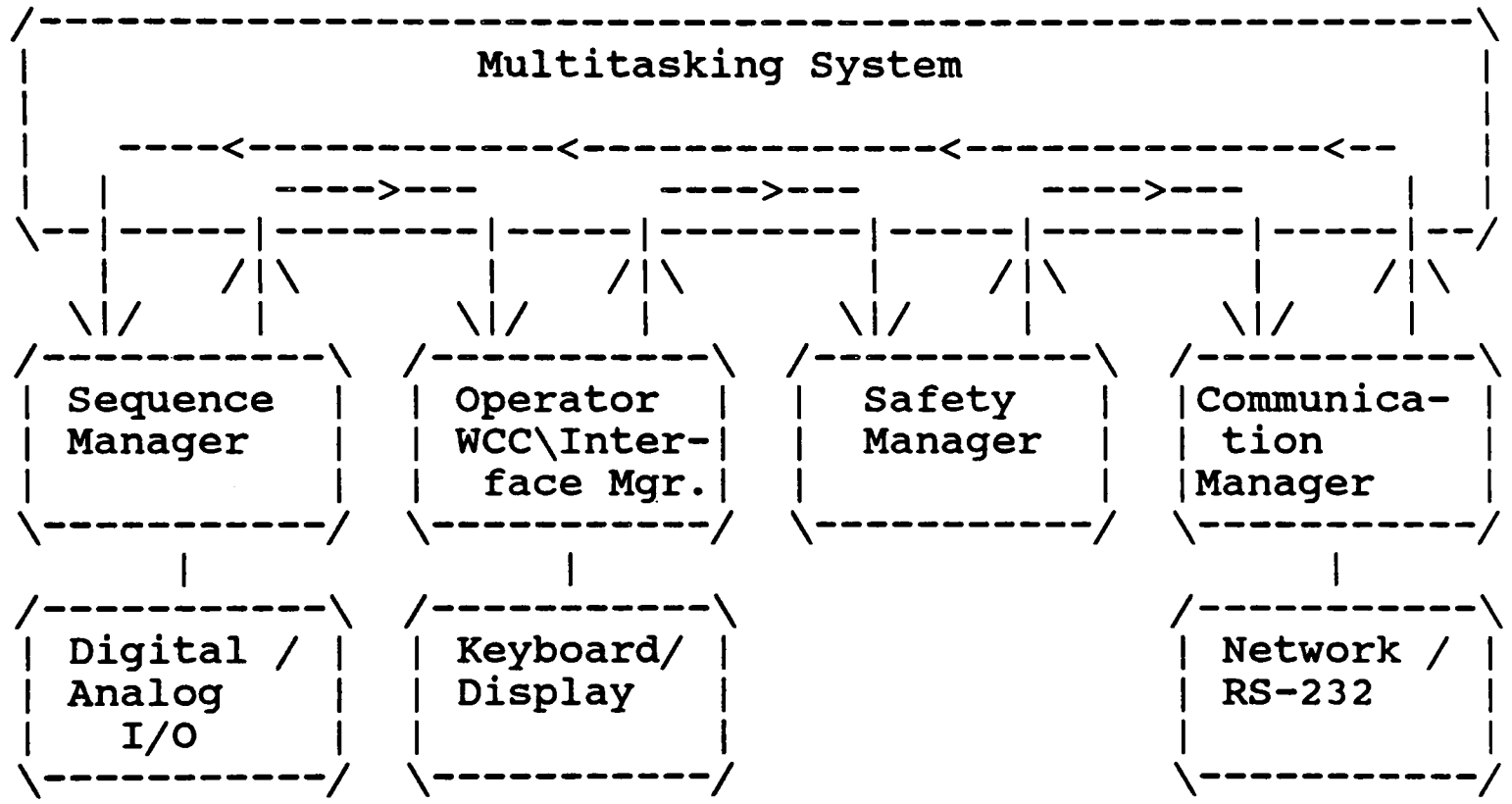


Figure 4.1 Multitasking Software Structure

This multitasking system simply allocates short periods of time or time slices of the Central Processing Unit (CPU) to the individual tasks in the task list. At the end of the time slice, the task is temporarily suspended or preempted and another task is given control of the CPU, which in turn runs for a short time until it is preempted.

There are two methods used to determine when a task gives up control of the CPU and another task takes control. One method is referred to as "preemptive" mode and the

other is "nonpreemptive" mode. Both methods have advantages and disadvantages. Preemptive mode places the responsibility of task transitions on the multitasking system. The task transitions will occur when a timer, kept by the time slicing program, expires. In preemptive mode, a task can be suspended and another started at any point during the task; that is, task transitions or context switches are asynchronous to the running tasks. The tasks are not aware of the context switches which can occur at any point in the program. Preemptive mode is most useful when task synchronization is not a concern and tasks are, for the most part, independent of each other. If tasks are not independent or if the tasks share common resources, then a method is needed to prevent possible conflicts. Sections of the code will have to be marked by the programmer as "critical," if the interruption to that section could cause conflicts. The multitasking system will not allow a task access to critical code if some other task has started but not completed a critical section.

Nonpreemptive operation requires the task itself to release control of the CPU by executing a "give up control" statement, which places the responsibility on the programmer for efficient task transitions [15]. Nonpreemptive operation is advantageous because context switches can be controlled, which reduces task synchronization problems. Due to potential synchronization problems between the

various managers in this system, nonpreemptive operation was selected. As indicated in Figure 4.1, the Sequence Manager, Safety Manager, Operator Manager, and the Communication Manager are allowed access to the CPU by the multitasking system. Each task must give up control of the CPU immediately after completing one pass through the appropriate code sections. The use of a multitasking software structure simplifies programming since individual tasks can be developed somewhat independently.

To ensure adequately fast response to input changes and operator keyboard inputs, these tasks must be serviced by the CPU at a rate of at least 10 times per second. If the system is run in the nonpreemptive mode, each task must give up control at an appropriate time and point in the program. Task synchronization and code section execution times must be considered when planning the locations for task swaps. For example, the Operator Manager is normally only scanning the keyboard and updating the active status screen. After making a single pass through these short routines, control of the CPU is relinquished so that other tasks may have control of the CPU. If the preemptive mode were used, most of the time allocated to the Operator Manager task would be wasted, since the execution of the screen update and keyboard scans would normally take a small percent of the allotted time slice. The Sequence Manager

makes one pass through the sequence code and executes whatever needs to be done at the current time. Immediately after completing this single pass through the code, CPU control is given over to another task. Other tasks call only the routines necessary for the particular state that the machine happens to be in at the point when the task receives control of the CPU. For example, if there are no messages to transmit nor any messages received, the Communication Manager can give up control almost immediately; thereby, increasing the task scan rate.

Systems which have Proportional-Integral-Derivative (PID) loops may require higher scan rates. Some control applications have PID control loops which must be serviced every 100 milliseconds in order to provide the needed response. To adequately service the PID routines, the Sequence Manager must gain control of the CPU at least 20 times per second. For most control applications, a rate of 20 scans per second is sufficient to be considered a real-time control system.

Main Program

The Main Program of this control system is relatively simple since its basic purpose is to initialize program variables and data structures and initiate or spawn the four program tasks. In order to start the control program, it is necessary to enter a valid pass code. A software

protection key is also checked by the Main Program before equipment control is initiated. The following sections explain the operation of the Main Program in greater detail. Figure 4.2 shows the flow chart for the Main Program.

Introduction

The introduction to the Main Program provides the name of the equipment, name of the manufacturer, date of manufacturer and installation, software version number, a copyright notice, and perhaps other information or instructions. It is very important that the software version number be displayed since many possible versions could exist. The introduction screen also is important because it instructs the user how to initiate the control program.

Pass Code

To prevent equipment access by those who are not authorized, a valid pass code entry is required prior to equipment initialization. If a valid pass code is not entered, the program will terminate. Valid pass codes are stored in a data table which is loaded before the pass code check. Users with the correct pass code and associated access level may view and update the list of valid pass codes. A pass code editor routine provides this feature.

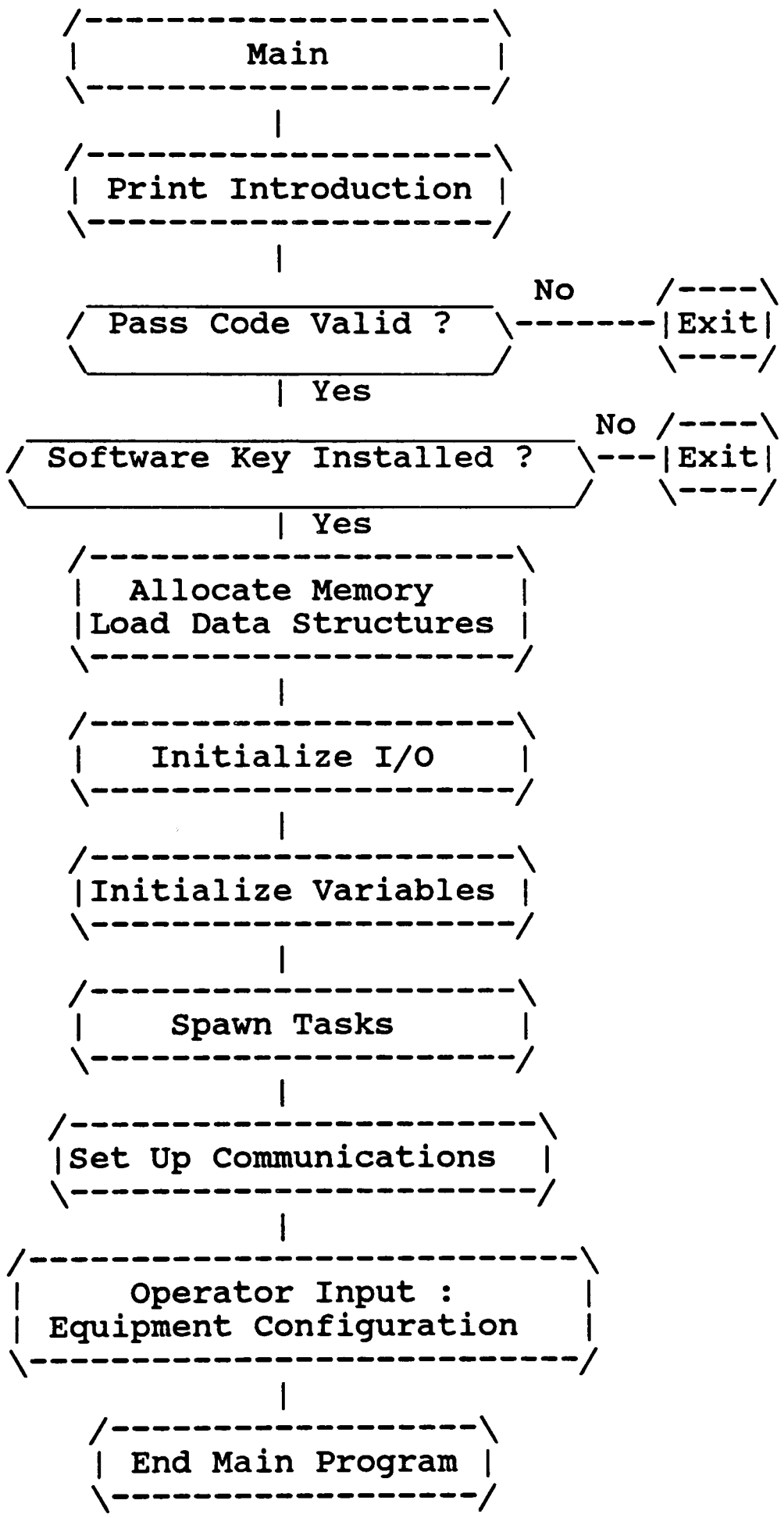


Figure 4.2 Flow Chart of Main Program

Software Key

It is important that some effort be made to prevent others from stealing the design of the equipment and its associated control software. Since it is difficult to prevent the duplication of the equipment itself, a software method of protection is implemented. The software is not copy protected; however, a software key is checked by the control program before equipment access is granted. This customized key can only be obtained by the original equipment manufacturer. The control program sends a string to the key and the key returns a unique code, which is a function of the special customized key and the string the key received. Other keys will not return the same response even with identical inputs.

The character string sent and the corresponding key replies can be stored in the data tables. Access to this table editor should be allowed only by the equipment vendor. In this way, the vendors can protect their software from being used without a custom key. These data files would be stored in a coded format to prevent easy deciphering of the key codes.

This protection provided by the software key is not infallible, but it does provide some level of protection. To increase the level of protection, this control program makes software key checks at several places in the program, in addition to the check during initialization.

Allocate Memory

For this software system, it is necessary to allocate memory to provide a dedicated memory location for the generic data tables used by the code. Once the memory blocks are allocated, the data tables can be transferred from the disk files to the dedicated memory locations.

Load Data Structures

The generic source code relies on specific control data in order to provide the information needed to control the equipment. These data tables are stored as disk files and must be loaded into the control program's memory before equipment control can begin. These data files are copied into the appropriate allocated memory locations during the loading operation.

Initialize I/O

Before the equipment can be controlled, the I/O interface has to be verified and initialized. All digital and analog outputs have to be set to the safe or idle state, which is normally "ZERO" output. The safe state for the equipment is designed to be all control outputs set to "ZERO," to provide for a safe state in the event of a power failure.

Initialize Variables

Various program variables, constants, flags, and counters have to be assigned initial values before the program can begin. Many of these variables are a part of the application specific code and are not initialized during the Load Data Structure operation.

Spawn Tasks

The four tasks, Sequence Manager, Safety Manager, Operator Manager, and Communication Manager are initiated or spawned by the multitasking operating system. These four subprograms or tasks will share access to the CPU as described in the section on multitasking in this chapter.

Communication Configuration

The communication ports must be verified for correct operation and initialized so that communications can be established between the control computer and the slave subsystems. In this system, smart stepper motor controllers are connected to the RS-232 port of the computer. This system has no WCC Interface; however, if one were installed, initialization of this network interface would also have to be done.

Equipment Configuration

The Operator Manager would have to allow the user to set up the equipment control program for the specific

application. It may be possible that the equipment could operate in several different modes. These different modes could be selected from the Configuration Menu before equipment initialization is completed. This system provides a data table with equipment configuration parameters on a disk file; however, this file is specific to this application.

End Main Program

The Main Program ends, but the tasks spawned earlier are still running. These tasks will run until the program is terminated by the user. The program cannot be terminated if the machine is running. Program termination can only be accomplished if the equipment is idle and in a safe state. Upon program termination, all outputs are turned off, allocated memory is freed, and other routines are executed to restore the system to its original state.

Sequence Manager

Description of Data Structures

The purpose of the Sequence Manager is to control the operation of the programmable sequencer or finite state machine so that the correct sequence can be executed at the proper time. The following sections describe the various data structures used by the generic Sequence Manager.

Queue table. The operation of the Sequence Manager is centered around a dynamic data table referred to as the

Queue Table. This table contains all the information needed by the Sequence Manager to determine the exact state of the program and the equipment. Information such as the active sequence number, step within that sequence, timer values, counter values, and flags are maintained by the Sequence Manager in the Queue Table. The data contained in this table provide a "snap-shot" of the entire machine status at any point in time. The Sequence Manager uses this table to keep up with the equipment status so that sequential operations can be carried out. This table performs the task of the "memory" block of the finite state machine analogy described in the previous chapter. The state of the equipment or of any finite state machine is determined by the history of its inputs. The data in the Queue Table that is used by the generic program sections is listed in Table 4.1. Since this software system controls four modules, there are four copies of each table entry, one for each module. There are many other Queue Table entries that are not a part of the generic code and are omitted from Table 4.1. These Queue Table entries deal with the application specific code. Special timers, flags, equipment status variables, process status variables, and maintenance status variables make up the remainder of the Queue Table.

Table 4.1 Queue Table

<u>State Data</u>	
Sequence Number	Sequence Reference No.
Step Number	Step or State No. in the Referenced Sequence
<u>Flags</u>	
Sequence Fault Flag	Fault Due to State Input Conditions Not True by Expiration of Time-Out
Safety Manager Fault Flag	Fault Detected by Safety Manager
<u>Timers</u>	
Time Delay (Timer)	General Purpose Timer
Time Out (Timer)	Timer Used to Detect Sequence Faults
Step End Time (Timer)	Timer Used to Mark the End of a Timed Step

<u>Process table</u>. The functionality of the "next state decoder" of the finite state machine is provided by the Process Table and the Digital and Analog Sequence Table. The Process Table, Table 4.2, contains a list of sequences that must be executed in order to complete a certain process or maintenance operation. For example, the normal order of sequences for a certain process equipment is to complete ModLoad (Module Load), ChamLoad (Chamber Load), Etch, ChmUnload (Chamber Unload), and ModUnload (Module Unload). Other applications of this generic control system would require a different sequence list, that is, a list suited to the specific application.

Table 4.2 Process Table

	Module 1	Module 2	Module 3	Module 4
Process Sequence Names	ModLoad	ModLoad	ModLoad	ModLoad
	ChamLoad	PreRinse	PreRinse	ChamLoad
	Etch	ChamLoad	ChamLoad	Etch
	ChmUnload	Rinse	Rinse	ChmUnload
	ModUnload	ChmUnload	ChmUnload	ModUnload
		ModUnload	ModUnload	

Digital Sequence Table. State transition data for each sequence listed in the Process Table is contained in the sequence tables. The Digital Sequence Table, Table 4.3, contains the input conditions required to initiate a state transition and output values that are to be set at the time of a state transition. As can be seen in the example data in Table 4.3, Step 0 has no inputs and the DrCl (Door Close) output is asserted. In Step 1, the DrCl (Door Close) input is checked to be True and the DrOp (Door Open) input is checked to be False. If these two input conditions are valid before the time-out associated with the DrCl output expires, the outputs for Step 1 are set and a state transition to Step 2 occurs.

Table 4.3 Digital Sequence Table

Sequence Name - Etch

STEP	N_DIS	TRUE	FALSE	N_DOS	ASSERTED	NOT ASSERTED
0	0			1	(0)DrCl	
1	2	(0)DrCl	(1)DrOp	1	(1)V-1	
2	2	(0)DrCl	(1)DrOp	1	(2)V-2	
3	2	(0)DrCl	(1)DrOp	2		(1)V-1 (2)V-2
.	.	...	...	.	...	...
19	.	...	...	.	...	...

Nomenclature :

STEP	- Sequence STEP Number(0-19)
N_DIS	- Number of Digital InputS
TRUE	- Input Condition TRUE
FALSE	- Input Condition FALSE
N_DOS	- Number of Digital OutputS
ASSERTED	- Output ASSERTED
NOT-ASSERTED	- Output NOT ASSERTED
(#)	- Pointer to Digital Hardware Address

Analog Sequence Table. The Analog Sequence Table, Table 4.4, contains analog input and output conditions that are used by the Sequence Manger to control the state of the machine. Table 4.4 also contains step times which can be used by the Sequence Manager to initiate state transitions when state changes are driven by time.

Table 4.4 Analog Sequence Table

Sequence Name - Etch

STEP	N_AIS	AIC	IN VALUE	N_AOS	AOC	OUT VALUE	TIME
0	1	1	100 (N2)	1	1	100 (N2)	1.0
1	2	1	80 (N2)	1	1	80 (N2)	2.0
		3	50 (HF)		3	50 (HF)	
2	2	1	70 (N2)	2	1	70 (N2)	5.0
		3	20 (HF)		3	20 (HF)	
.	.	.	..	.	.	..	.
19	.	.	..	.	.	..	.

Nomenclature :

STEP - Sequence STEP Number (0-19)
N_AIS - Number of Analog InputS for this Step
AIC - Analog Input Code (References the Analog Hardware Address)
IN VALUE - INput VALUE Desired
N_AOS - Number of Analog OutputS for this Step
AOC - Analog Output Code (References the Analog Hardware Address)
OUT VALUE - OUTput VALUE
TIME - Step TIME in Seconds

Digital Address Table. The "output decoder" section of the finite state machine is handled by the hardware address reference tables. These tables are used in conjunction with the sequence tables to reference the desired digital and analog inputs and outputs. The Digital I/O Hardware Address Table is shown in Table 4.5 with some example data.

Table 4.5 Digital I/O Hardware Address Table

Module 1 - ETCH

InCode	Name	Byte	Bit	Fault Code
0	...	.	.	.
1	DrOp (M1)	4	0	1
2	DrCl (M1)	4	1	1
3	...	.	.	.
.	...	.	.	.
23	...	.	.	.

OutCode	Name	Byte	Bit	Time Out
0	...	.	.	.
1	DrOp (M1)	0	0	3.0
2	DrCl (M1)	0	1	3.0
3	Rel-1(M1)	1	7	0.0
.	...	.	.	.
23	...	.	.	.

Module 2 - RINSE

InCode	Name	Byte	Bit	Fault Code
0	...	.	.	.
1	DrOp (M2)	4	6	1
2	DrCl (M2)	4	7	1
3	...	.	.	.
.	...	.	.	.
23	...	.	.	.

OutCode	Name	Byte	Bit	Time Out
0	...	.	.	.
1	DrOp (M1)	5	0	3.0
2	DrCl (M1)	5	1	3.0
3	Rel-1(M1)	6	7	0.0
.	...	.	.	.
23	...	.	.	.

(Module 3 and Module 4 contain similar data)

This table maps the data in the Digital Sequence Table to the actual digital hardware addresses. Pointers are used in the Digital Sequence Table to reference the digital input and output addresses. Actual addresses could have been stored in the sequence tables, but since this software system allows for multiple modules, a separate sequence table would have to be allocated for each module. Many of the sequences are the same for each module, with the only difference being the hardware addresses. In order to simplify the program structure, an indirect address referencing scheme is used to eliminate the coupling between the Digital Sequence Table and the hardware addresses of the individual module.

Digital I/O access. In order to reference an individual digital input or output, the base address of the I/O system must be known. If only one I/O interface is used the source code can assume that all I/O are referenced from a single base address, which can be stored in a configuration data structure. A system which requires more than one I/O interface will also require an equal number of I/O base addresses to provide access to all the inputs and outputs.

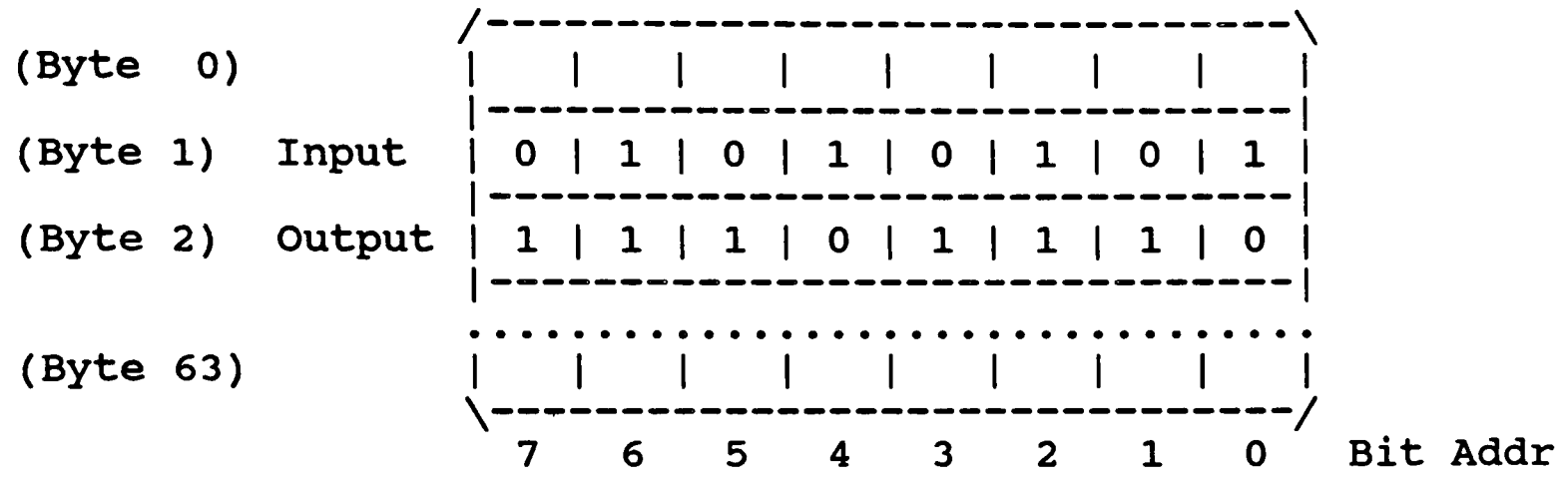
With the base address known, the individual digital inputs and outputs can be referenced if the byte and bit offsets from the base address are known. It is these offsets that are stored in the Digital Hardware I/O Table. The byte offset is the byte number within the I/O address

space where the particular input or output bit is assigned. The bit number is the offset within the particular byte where the desired input or output resides. In order to access a desired input bit, the entire byte where the input resides is read. Once the byte containing the desired bit is read, the bit address is used to generate a mask to determine the value of the input bit.

Digital outputs are set by using a mask which is determined by the bit offset. If a logic one is to be set, the value of the byte containing the desired output bit is "or-ed" with a mask containing a "one" in the bit position referenced by the bit offset. Zeros fill the other mask bit positions. If a logic zero is to be set, the value of the byte containing the desired output bit is "and-ed" with a mask containing all ones, except with a "zero" in the bit position referenced by the bit offset. This generic method of digital hardware I/O referencing functions well for this particular I/O system. Examples of digital input and output operations are given in Figure 4.3

Analog Address Table. Analog I/O hardware addressing is done in a similar way to the digital addressing. The Analog Sequence Table, Table 4.4, contains pointers AIC and AOC that point to the Analog Hardware Address Table shown in Table 4.6.

I/O Address Map



Example Bit Read Operation :

```
Input Byte Address = 1
Input Bit  Address = 2

Input Bit Value  =   Input byte     AND     Bit Mask

0 0 0 0 0 1 0 0  =  0 1 0 1 0 1 0 1  &&   0 0 0 0 0 1 0 0

IF (Input Bit Value > 0)

   THEN  Input Bit Value = "ONE"
   ELSE  Input Bit Value = "ZERO"
```

Example Bit Write Operation : (Set Bit to "ONE")

```
Output Byte Address = 2
Output Bit  Address = 4

New Output Byte  =  Old Output Byte   OR       Mask

1 1 1 1 1 1 1 0  =  1 1 1 0 1 1 1 0   OR   0 0 0 1 0 0 0 0
```

Example Bit Write Operation : (Set Bit to "ZERO")

```
Output Byte Address = 2
Output Bit  Address = 7

New Output Byte  =  Old Output Byte  AND   Inverted Mask

0 1 1 0 1 1 1 0  =  1 1 1 0 1 1 1 0  AND   0 1 1 1 1 1 1 1
```

Figure 4.3 Example I/O Operations

Table 4.6 Analog Hardware Address Table

Module 1 - Etch

Analog Code	Analog Name	IN_ADDR (Byte)	IN_CHAN (Number)	OUT_ADDR (Byte)
0	N2_MFC (M1)	0x10	0	0x20
1	Vapor_MFC (M1)	0x10	1	0x21
2	HF1-MFC (M1)	0x10	2	0x22
.	...	.	.	.
19	...	.	.	.

Module 2 - Rinse

Analog Code	Analog Name	IN_ADDR (Byte)	IN_CHAN (Number)	OUT_ADDR (Byte)
0	Spin Motor(M2)	0x10	15	0x27
1	N2_MFC (M2)	0x10	14	0x26
2	...	.	.	.
.	...	.	.	.
19	...	.	.	.

(Module 3 and Module 4 contain similar data.)

The Analog Sequence Table contains pointers that reference the analog address data by use of the "Analog Code," or AIC for analog inputs and AOC for analog outputs. Referencing analog outputs requires only the byte address and the I/O system base address since an analog output uses eight bits of I/O address space. Analog inputs require the I/O system base address, the byte address of the analog input board, and the analog channel number, zero to 15.

Flow Chart Explanation

The main program of the Sequence Manager is simple since only a few routines are called from its main program. The complexity of this task is embedded in the routines called from the Sequence Manager. The flow chart for the Sequence Manager main is shown in Figure 4.4.

Calculate tic time. A continually increasing "tic" timer is needed in order to make delta time measurements. The MS-DOS operating system updates certain registers that contain the year, month, day, hour, minute, second and hundredths of seconds. Some of these values are used to generate a linearly increasing tic timer and the resulting tic count is used to make delta time measurements. Tic time is kept in tenths of seconds from the time the system was started. This number can grow quite large; therefore, it is stored as a 32-bit number.

Loop counter. The Sequence Manager must scan all modules in the equipment system; therefore, a loop is set up to allow for all modules to be serviced. The module number, M, is used to reference the data associated with the specific modules. Once all modules are serviced, the Sequence Manager gives up control of the CPU. When control is regained, the Sequence Manager loop is repeated.

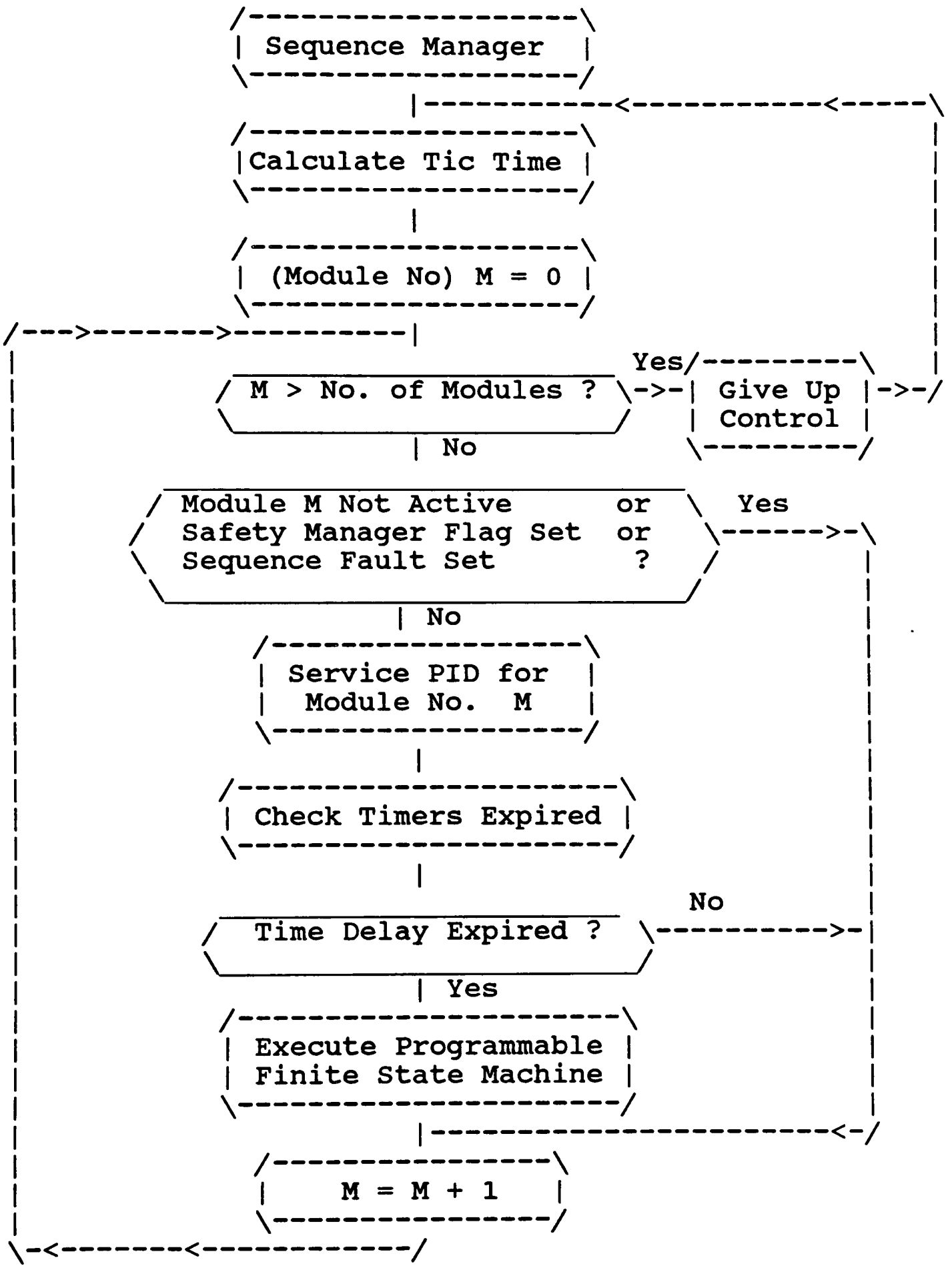


Figure 4.4 Flow Chart of "Sequence Manager"

Bypass conditions. If the module is not active or if there is a Safety Manager fault condition or if a sequence fault has occurred, then the Sequence Manager bypasses all of the service routines for that module.

Service PID loop. The PID routine for each module must be called each time the Sequence Manager gains access to the CPU. The PID routine may simply return to the calling program if its period timer has not expired or if the PID control loop has been disabled by some previous event. This PID control section has some generic features. The PID constants; that is, the proportional, integral, derivative, and PID timer or period constants are stored in a data table that can be edited.

Check timers expired. As mentioned before, certain maintenance or calibration routines have to be run every so many hours or number of machine cycles. For example, a process module has to initiate a maintenance routine every six hours. After the six-hour maintenance timer expires, the Sequence Manager will run the maintenance routine. In come cases, the execution of the maintenance routine may have to wait for an operator or WCC acknowledgment. The number of hours between runs can be set up in the module configuration table.

Execute finite state machine. The programmable finite state machine is called each time the Sequence Manager gains access to the CPU if the above conditions are

satisfied and if the Time Delay timer has expired. Often, there is nothing for the programmable sequencer to do since the task scan rate is many times faster that the response time of the equipment. In other words, to carry out the sequential operations, it may be necessary to initiate some action every few seconds. Since the task scan rate is much greater than what is normally required, this programmable finite state routine can be bypassed most of the time; thereby, saving CPU time for other tasks. As can be observed from the flow chart in Figure 4.4, any of the four conditions will prevent the execution of the programmable sequencer. These conditions are, a module that has not been initialized, a previously set time delay has not expired, a sequence fault has been detected, or a module fault has been detected by the Safety Manager. If all of these conditions are false, the active sequence is executed. The active sequence is referenced by the data in the Queue Table, which contains the present state of the finite state machine.

The flow chart, Figure 4.5, shows the operation of the programmable finite state machine. This routine is responsible for the execution of the sequential operations.

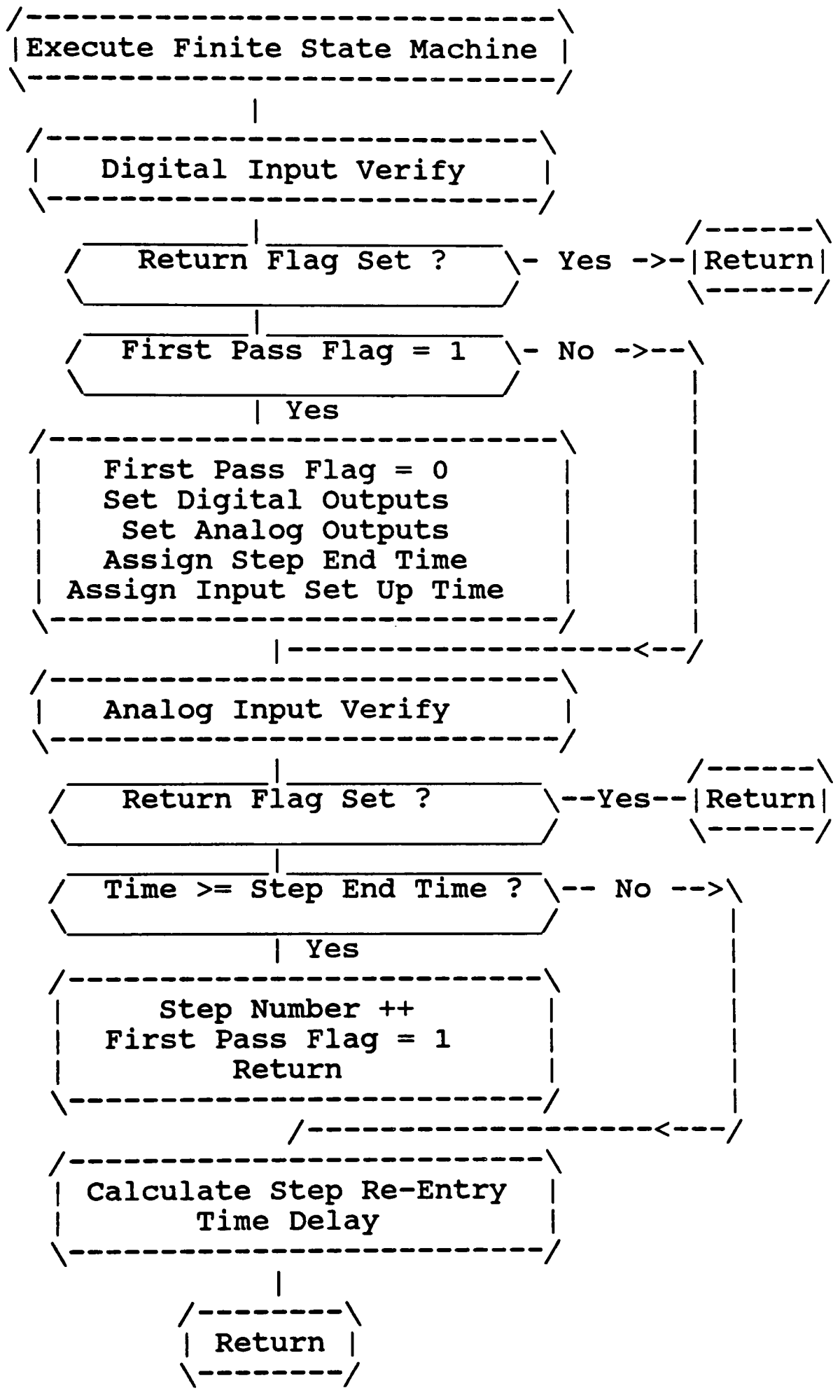


Figure 4.5 Flow Chart of "Execute Finite State Machine"

Digital input verify. Each state or step in the sequence may have certain digital input conditions which must be satisfied before the outputs for that state can be set. From Figure 4.6, it can be seen that valid input conditions will cause "Digital Input Verify" to return to "Execute Finite State Machine" ("EFSM") with a Return Flag of "0," which enables the continuation of "EFSM." If the input conditions are not satisfied upon entry into the state, several possible actions will result. The first condition checked is the Time-Out timer. The Time-Out represents the time in tics when the action that set the Time-Out should be completed. If the timer has not expired and the digital inputs are not valid, "Digital Input Verify" returns with a Return Flag of "1," which invokes an immediate return from "EFSM." The next time that "EFSM" and "Digital Input Verify" are called, the same inputs and Time-Out timer will be checked again. This process will continue until the input conditions for the step are satisfied or the timer expires. If the timer expires before the input conditions are valid, a fault code associated with the faulted input determines the next action; however, if the input conditions become valid before the timer expires, sequential operations continues as expected.

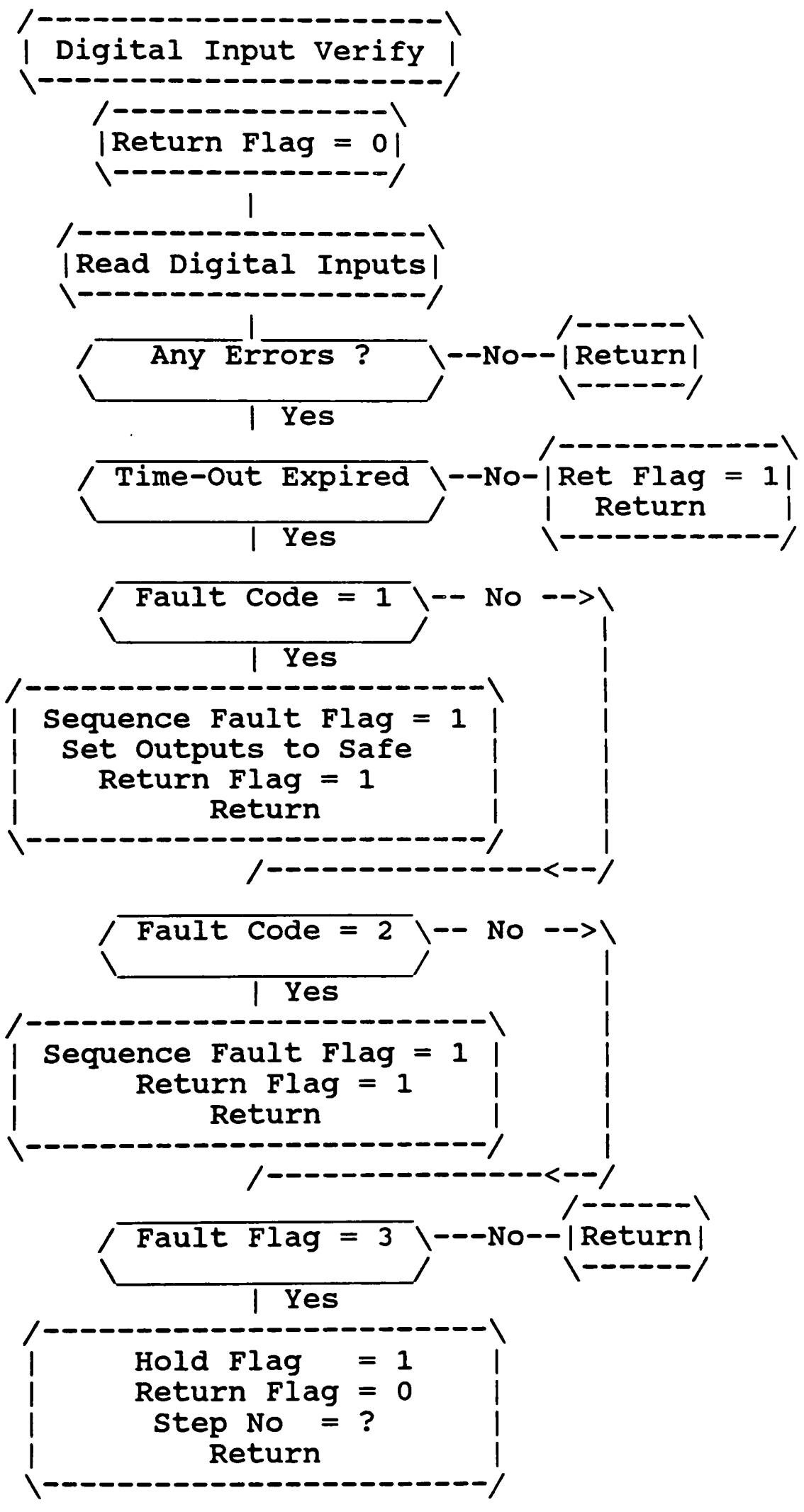


Figure 4.6 Flow Chart of "Digital Input Verify"

The occurrence of a priority one fault results in the setting of the sequence fault flag along with the setting of the outputs to the safe state. A priority one fault also prevents subsequent calls to "EFSM," until the fault is acknowledged by the operator and the fault cleared. All module outputs are set to the safe state and the return flag of "1" is set so that the remainder of "EFSM" will not be executed. Priority two faults are less serious and require only that the sequence be suspended by the setting of the sequence fault flag. Priority three faults do not require that the sequence be suspended at the time of the fault. The setting of the hold flag will cause the sequence to be suspended in a more appropriate step later in the sequence. Priority three faults may also initiate a change of the step number in order to bypass a part of the sequence.

First pass flag. Referring to Figure 4.6, The first pass flag has a value of "ONE" upon the first entry to the step. During this first pass, the digital and analog outputs are set along with the two timers, if the digital input conditions were correct. The Step End Time is read from the Analog Sequence Table and is converted from seconds to the future tic time when the step is to end. This operation is done by adding the table value in tics to the present time in tics. The resulting value is the future time when the step is to be completed. The "Step Set Up

Time" is also read and converted to a tic count. This timer tells the "Analog Input Verify" routine when the analog inputs should be valid.

Analog input verify. The analog inputs that were set during the first pass of the step should be checked after a delay determined by the "Step Set Up Time" timer. This routine functions similar to "Digital Input Verify" except it reads the Analog Sequence Table instead of the Digital Sequence Table. To detect an analog fault, the input value must be outside the programmable window. For example, an error window could be read from the data table or an error window could be calculated to be a certain percent of the desired input value. If a fault is detected, similar action would be taken as in "Digital Input Verify."

Step end time check. Each pass through "EFSM" requires a check for the end of the step. If the present tic time is greater or equal to the "Step End Time," the step number is incremented, the first pass flag is reset, and a return to the main Sequence Manager routine is executed. If the end of the step has not occurred, a step re-entry time is calculated. This re-entry time will be one of two values. In most cases, the re-entry time is the present time plus the sequence scan rate, which is one second for this system. If there is less that one second remaining in the step the next entry time is simply the step end time.

Safety Manager

The responsibility of the Safety Manager is to ensure that operator, material, facility, and process equipment are not subject to dangerous conditions. If such a condition occurs, the system will take an appropriate action to put the system into a safe state. In order to provide adequate response time, the Safety Manager must be serviced or scanned, for many systems, at a rate greater than 10 times per second. The Safety Manager should prevent operation of the machine in unsafe ways, not execute invalid keyboard or WCC commands, and provide fault recovery routines for all possible fault conditions.

These requirements make the Safety Manager quite difficult to design since it must provide safe operation under an almost infinite set of circumstances. In order to simplify the design, safety objectives were divided into the following classes.

Fault Classifications

Priority one faults. The first and highest level of safety concern is referred to as a "priority one" fault condition. A priority one fault requires that all operation must be suspended, all outputs must be set to a safe state, a message describing the fault is sent to the user interface, and no additional action can take place until the fault is corrected. Examples of priority one faults

are the detection of HF gas, loss of facility N2 supply, or an operator initiated Stop. Faults classified as priority one must be handled as described to prevent unsafe conditions from existing.

Priority two faults. Priority two faults deal with the sequential operation of the equipment. In order to safely execute a sequence, certain input conditions have to be valid before the next step can be started. If the required input conditions for the execution of a step are not valid with in a certain time, a priority two fault occurs. This fault condition is also called a sequence fault, because the sequence can not continue until the fault is corrected. These faults are not serious and present no danger to the operator or equipment. The detection of these faults, however, prevents dangerous condition from occurring. For example, during a Chamber Load sequence, the chamber open sensor is checked before an attempt is made to load the chamber. If the chamber open is not detected, a priority two fault is set and the sequence is suspended. This protection prevents the equipment from making a chamber load attempt with the chamber not open; therefore, avoiding possible equipment or material damage. Sequential operation can continue once the fault is corrected.

Priority three faults. Some faults that occur do not require any immediate action; however, an appropriate

action may be required at a later time. This level of fault will result in a message being sent to the user and a flag set to suspend sequential operation at a later time. At the appropriate step in the sequence, operation will be suspended until the fault is corrected. These faults are less serious, but still require the attention of the user.

Operator faults. Safe operation requires that the user or WCC not be allowed to command the machine to do something that could cause a dangerous condition. For example, the maintenance user could not open a HF source valve without the chamber closed. If he were allowed to open the valve with the chamber open, HF gas could be vented into the room.

Another safety concern is the inputting of data. Only valid data should be accepted from the keyboard or WCC. Obviously, the system cannot distinguish between all data inputs as to the validity; however, there are some higher and lower limits that input data values must be within. The input data is checked to be within the limits after it is inputted. Input data outside the allowed ranged is ignored and a message is sent to the user screen. By preventing invalid or unsafe data from entering the system, many safety and process problems can be avoided. An example of this level of protection is as follows. The maximum number of wafers in a carrier is 25. If the user

attempts to enter 30 wafers, the input is ignored. If this feature were not built in, the wafer transfer robot would run into its upper mechanical limit in an attempt to access wafer positions outside its range.

Safety Manager Operation

The Safety Manager scans the Safety Table, shown in Table 4.7, looking for unsafe conditions. If such a condition is found, the corresponding fault code determines the action that will be taken. The flow chart in Figure 4.7 describes the operation of the Safety Manager.

Table 4.7 Safety Table

Input Ref.	Name	Time Delay	Fault Value	Fault Code
0	Stop Button	0.0	1	1
1	N2 Supply	1.0	0	1
2	Water Temp	2.0	1	3
3	HF Detect	0.2	1	1
.	...	...	...	...

The Safety Manager scans all the inputs that are in the Safety Table. When an input is first noticed to be in a faulted state, a fault is not immediately assigned to the input. The input has to remain in the faulted state for a certain time, which is specified by the System Time Delay (S_Time_Delay). This time delay prevents short mechanical and electrical glitches from triggering the safety

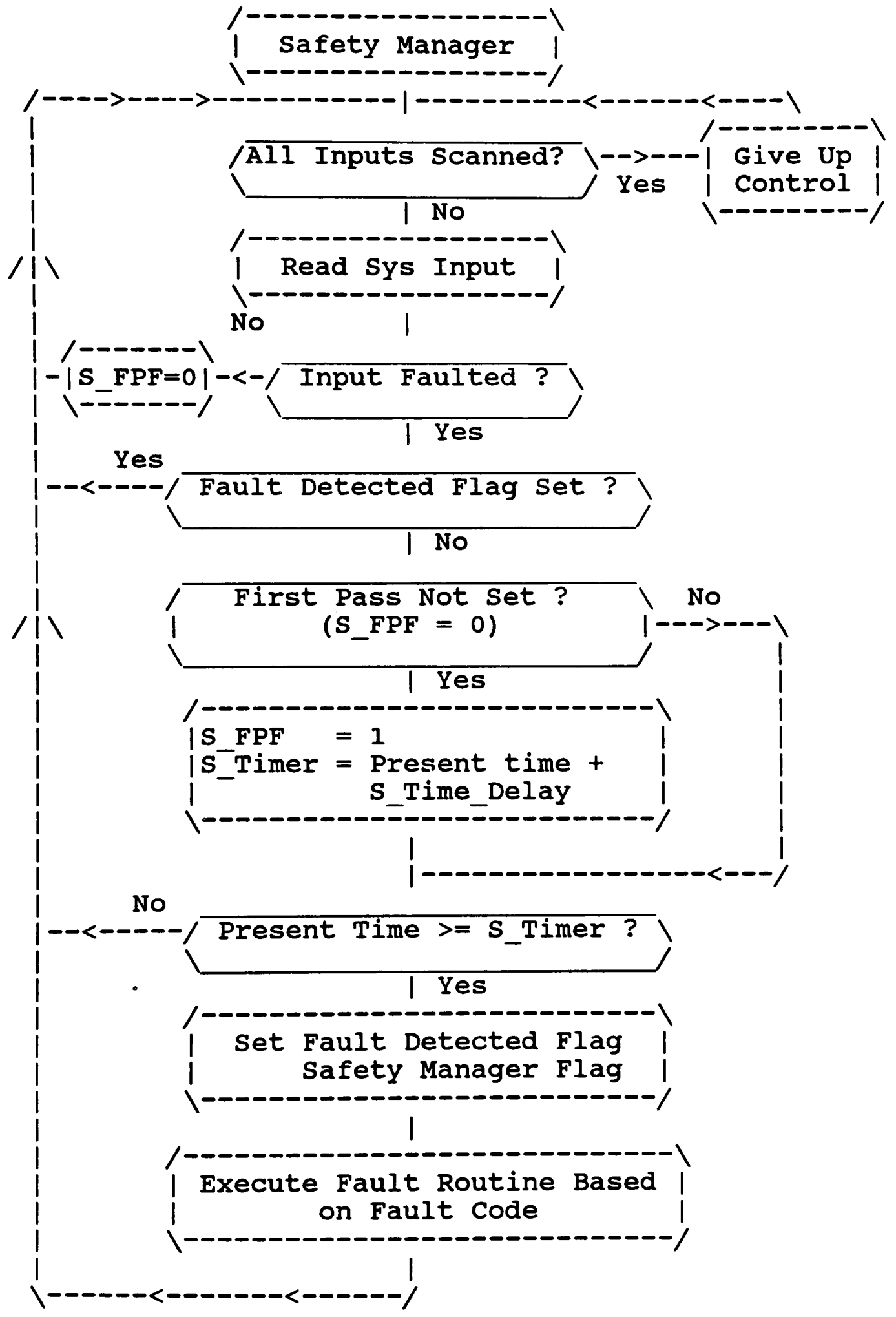


Figure 4.7 Flow Chart of "Safety Manager"

routines. Only after the input has remained in a faulted state for the time delay period, is it considered to be faulted. This time delay is programmable.

Each system input also has a fault code associated with it, which is used to reference the correct fault routine in the event of a fault. Once the input has been at the fault value for the required time, the input's Fault Detected Flag is set. This flag marks the input as faulted for later reference, in the event that the input becomes not faulted in the mean time. The detection of a priority one fault also sets a flag that suspends the operation of the Sequence Manager.

Software Structure Summary

The objective of this software project was to develop a generic process equipment control program. The sections of the control program and its associated data structures discussed to this point make up the generic, table-driven, portion of the program. In order to prove the flexibility of generic process control, this software was applied to an actual application in the semiconductor industry.

Due to the complexity of the implementation, the generic control software was unable to provide all of the flexibility needed; therefore, some sections of the program were designed specifically for this application. The following chapter discusses these application specific

sections and the results of the application of generic control software to this particular implementation. A portion of the code is included in the Appendix.

CHAPTER V

DESIGNING AN APPLICATION

The following sections describe the method required to design an equipment control system for a particular application using the generic program. This discussion covers the programming of the system data tables that currently exist in the generic software.

High-Level State Diagram

The first step to designing any control application is to define the sequential functions of the equipment. A general state diagram must be designed that describes the function of the equipment at a high level of operation. For example, a certain equipment may require a sequence of steps to load it with material. This sequence of steps could be called "Load Equipment." Another sequence of steps could be "Process Material," while another could be "Unload Equipment." A flow chart must be designed utilizing these high-level sequence descriptions to describe the functioning of the equipment at a high level. Each of these sequences of steps is referred to as a "function". Once all the functions for an application are defined, the names can be entered, into the Process Table.

The Process Table specifies the state diagram of the equipment at the highest level. For this example, function one is "Load Equipment," function two is "Process Material," and function three is "Equipment Unload." Another example of the Process Table contents is presented in Table 4.2.

Detailed State Diagrams

The detailed state diagrams for each function listed in the Process Table must be designed. An example of a section of a state diagram can be seen in Figure 3.3. These state diagrams contain the specific information needed to sequence the equipment through its various states. For example, the state diagram for the function "Process Material" must be clearly defined in such a way that its entire operation could be described from its state diagram.

Complete Digital Hardware Address Tables

From the detailed state diagram, a list of all the digital and analog input and output points is generated. These I/O points are needed to complete the Digital and Analog Hardware Address Tables. These inputs and outputs must be assigned a descriptive name and an I/O address. The address provides the link between the sequence tables and the actual I/O device. A fault code is also entered with each digital input indicating the fault routine that is to be executed in the event the Sequence Manager detects a fault in that particular input. Digital outputs have an

associated timer, called time-out, which indicates the maximum time required for the output, once asserted, to complete its action. These data are entered, via the Data Table Editor, into the Digital Hardware Address Table. An example of a completed Digital Hardware Address Table can be seen in Figure 4.5.

The Analog Hardware Address Table is completed in a similar manor. Descriptive names are entered for analog I/O points. Addresses are also entered for each point. Analog inputs also require a channel number. Figure 4.5 shows an example Analog Hardware Address Table with various inputs and outputs.

Each digital and analog I/O point is referenced by a pointer or a reference code. This code is used in the individual Digital and Analog sequence Tables to reference the desired I/O point. The completion of the sequence tables requires the use of these hardware pointers to reference the digital or analog I/O points.

Digital Sequence Table

Data from each detailed state diagrams is transferred into the corresponding Digital Sequence Tables. There is one Digital Sequence Table for each function listed in the Process Table. These sequence tables allow a maximum of 20 steps to be programmed for each high level sequence. Starting with step number zero, sequence data is entered

into the table until all the steps for the sequence are programmed. The number of digital inputs and outputs must be input for each step of the sequence. On the input side of the table, the digital inputs are referenced by the codes in the Digital Hardware Reference Table. If a digital input is suppose to be "True" its reference pointer is located in the "True" column of the table, otherwise, the input reference is located in the "False" column. The output side is completed by inputting the output reference pointers in either the "Asserted" column, if the output point is to be set, or in the "Not Asserted" column if the output point is to be cleared. An example of this table is shown in Figure 4.3.

Analog Sequence Table

The Analog Sequence Table is completed in a similar way as is the Digital Sequence Table. Analog input references are inputted to indicate which analog signal is to be verified at a particular step. The desired input value is also entered in percent of full scale (0 to 100 %). The output side of the table contains the desired analog set point value, which is also entered in percent. The last column is programmed with the desired step time in seconds. An example is shown in Figure 4.4.

Sequence Manager Summary

These preceding data table entries provide the generic source code with the information needed for sequential control of the digital and analog I/O for the application system. Once the data tables are completed, the editor will store these tables as disk files which can loaded by the software during the initialization of the application program. The control sequence can be revised simply by using the editor program to revise the data tables.

Safety Table

To design the safety section of the application program the designer should make a list of all the static safety input names and input fault values. For example, an equipment may have a dangerous gas sensor which must always read "Zero," for safe operation. If this input ever reads "One," then a fault must be detected and an appropriate safety routine executed.

There is only one data table associated with the Safety Manager. The structure of the Safety Table can be seen in Table 4.7. The system safety can be programmed by entering the same input reference pointers as are used in the Digital and Analog Sequence Tables. The pointer references the particular digital or analog input placed into the Safety Table. The time entry indicates how long the input must be in the fault state before it is considered to be an

actual fault condition. The fault flag must be entered to indicate which fault routine is to be execute in the event that the particular input is detected as a fault. Input fault conditions are also entered for each system input. The value of the input in the faulted state is entered in the column labeled "fault value."

Summary

The generic portion of this control program allows for the programming of the data tables that control the sequential operation of the digital and analog Sequence Manager. Also, the static Safety Manager can be programmed by completing the Safety Table. Other programming functions must be accomplished with application specific code.

CHAPTER VI

GENERIC IMPLEMENTATION

The original objective of this software project was to design a control system that provides the flexibility to implement a wide variety of process control functions without having to revise the actual control program. To provide this level of flexibility, all process control variables would have to be incorporated into the generic data structures. This project proved to be quite complex; therefore, only the generic structures described previously have been developed. The remaining functionality for the test case was implemented with application specific code.

The system is controlled by a personal computer. Multitasking software permits the single electronic module to manage up to four different process modules at the same time [15]. The equipment is a four-module Hydrogen-Fluoride (HF) Vapor Etcher system, manufactured by FSI International. This product, named Excalibur, is an ambient-temperature vapor-etching process equipment which needs neither plasma nor vacuum for the removal of silicon dioxide [16].

Each module has 20 digital inputs, 20 digital outputs, eight analog inputs, and five analog outputs. There is

also a RS-232 serial output port that communicates with two smart stepper motor controllers in each of the four modules. These stepper motors manipulate the wafer transfer mechanisms. These robots move wafers to and from the process chamber. An operator interface allows the user to obtain the status of each module at any time since the displays are updated in real-time. The interface also allows the user to command the machine to execute processes, perform maintenance routines, view and update process recipes, obtain machine and material status, and obtain fault information. The Safety Manager is also incorporated and performs all the safety functions required to satisfy customer safety requirements.

An interface to a WCC has not been implemented. The WCC interface would provide the same functionality as the included Operator Interface; however, the communications would take place over a network instead of using the keyboard and display. If the communication network and network interface could handle the data throughput, the control program could report the complete status of the equipment to the WCC in real-time.

This entire system is functional and approximately 30 single and multiple module systems have bee installed in semicondutor processing facilities worldwide [17]. In addition, the generic sections of this control software have been adapted to control a plasma etcher [18], a

coater-developer [19], and a noncontaminating water heater [20]. The following section discusses the actual implementation.

Multitasking Structure

The multitasking system chosen for this application is TimeSlicer (TM), which operates under the Micro-Soft Disk Operating System (MS-DOS) and runs on an IBM type Personal Computer (PC). An IBM type PC system was chosen because of its adequate computational power, the built-in operator interface keyboard and screen, the availability of a wide variety of interface hardware, and due to its low cost. TimeSlicer was chosen because it provided the needed multitasking functionality at a low cost. The use of these particular systems has little to do with the concept of generic process equipment control. Ideally, this system design could be ported to most any hardware and software system. The original test version was implemented on a TI-990 computer system [21]. A block diagram of the software system operating under TimeSlicer was shown in Figure 4.1.

The TimeSlicer is a linkable library which provides multitasking and real-time functionality at the program application level. It is not a multitasking operating system; however, it could be used to generate one [22]. For this application, only a few of the available multitasking functions were used. The main functions used are the

commands "give up control" (GUC), "critical start" (CRITSTRT), and "critical end" (CRITEND). The GUC function initiates a task swap or a context switch which allows a task to relinquish control of the CPU to another task. CRITSTRT and CRITEND provide a way to mark critical code sections. If a task has started but not finished a critical section, then other tasks cannot begin a critical section until the first task completes the critical section. These commands prevent conflicts when several tasks have to share common resources or DOS functions.

I/O Interface

The I/O interface used is the "MetraBus" system provided by the MetraByte Cooperation. This I/O system has a limitation of 512 digital I/O or 64 analog outputs [23]. Most I/O system designs require less capability than this limitation imposes. If the I/O requirements were to exceed this limitation, additional MetraBus I/O boards could be installed. These additional I/O boards would require that the Digital and Analog Hardware Address Tables be revised to also include a board reference for each input or output point in the system.

Generic I/O Sections

Each of the sequence routines that is called by the programmable finite state machine contains both generic and

application specific code. The generic code and associated data structures take care of the digital and analog input and output conditions that are used by the Sequence Manager to cycle the machine from one state to the next. Application specific code is used to sequence the machine when the generic code does not provide the needed functionality. Inputs from the keyboard and slave subsystem communication port, certain counters, flags, and timers have to be handled with application specific code. Outputs to the display screen, commands to the slave subsystem communication port, the setting of flags, incrementing of counters, and the starting of several timers are also carried out with application specific code.

Analyses of the Generic Features

In order to meet software delivery deadlines, some of the control functions were implemented with the application-specific code. The generation of generic code would have delayed the delivery of the software to the customer. Since the test application program was developed for an actual product and not strictly as a research project, design trade offs had to be made.

The results of these trade offs were interesting. As with any significant software project, revisions are required to upgrade the system. The revisions included adding new process sequences, restructuring the Operator

Interface screens, and incorporating new and revised hardware. The revisions that dealt with the generic sections of the code proved to be quite simple to change since these revisions involved only data file editing. Other changes that involved the application-specific code required much more effort and time. The compromise made by implementing the application-specific code saved some development time at the beginning; however, more time was required to revise these sections during the software maintenance phase of the project.

Software Verification

Software verification was necessary to prove that the generic program structures would function properly. This test was carried out on the Excalibur process equipment application. Software test procedures were written to enable the testing and execution of every branch in the program. All possible equipment functions and user input combinations were tested. The equipment was allowed to run in a looping mode for extended periods of time to test for time dependent errors.

In order to provide the most through testing, several people tested the functions of the program in addition to the software designers. Those who were familiar with the equipment and others who were not performed tests. By using various people to test the software, many bugs were

detected before the software was released to the customer. After the software was released, few real bugs were discovered by the customers. These extensive test procedures proved to be successful.

It can be observed from the test implementation that generic control software has the flexibility to solve many of the control problems that exist today. The following chapter will discuss this idea in greater detail.

CHAPTER VII

CONCLUSIONS AND RECOMMENDATIONS

In order to provide manufacturers with the level of process control that will be required in the future, equipment vendors will be required to develop more sophisticated control systems. This increase in sophistication will result in expensive and complex systems. A detailed top-down facility design plan is needed for the purpose of developing a frame work from which to build automated systems. With such a well documented top-down design, equipment vendors and manufactures will have a better understanding of what is required of automation equipment.

The goal of the top-down automation system design is to segregate the levels of automation into distinct groups and to define the requirements of each level of automation. These requirements are defined, so that future expansion can be made without having to redesign the control systems at the lower levels. Profitability results when there is provision to easily connect facility subsystems into work cells and when there is sufficient flexibility to accommodate new or revised operations [5].

In addition to the top-down plan, a generic process equipment control system could also provide much

flexibility in the implementation of automated systems at the equipment level. Since the design requirements for most process equipment are related, a generic, general-purpose, equipment control system can be designed. Such a general-purpose control system would provide a much needed tool for equipment designers to simplify flexible equipment control systems.

A control system has been designed and implemented that has many of the generic characteristics required. The generic portions of this control system are easily modified to accommodate for process or equipment revisions.

There is obviously a trade off in the attempt to obtain a more generic control package. A generic control system requires much more time and effort to develop than an application-specific control system. The first requires the development of generic data structures, control code, and data table editors. The data structures have to be flexible so that they provide the level of control required by most systems. The control code is made up of sets of procedures that operate on the generic data structures to carry out the desired operation. In addition, a data table editor has to be designed for the purpose of editing the data tables. Revision of the basic data structures means that both the control procedures and data table editor sections have to be revised.

Once a generic control code and support tools are developed, a significant reduction in the time required to develop a control system is realized. The time saved will result in a major reduction in the overall system development expense. Given a powerful generic control system, individual process equipment control systems can be designed much more quickly than previously. Another advantage is that the inevitable future equipment and process control revisions can be quickly and easily made by simply modifying the data tables instead of revising the control software itself.

Another distinct advantage of the generic control system is that the original software designers need not be consulted for the purpose of upgrading the application control system software. Using the generic system support tools, such as the data table editors, the process control engineers can revise the control program to suit the changing requirements. By reducing the dependency on the original software designers, future revisions are less likely to be hampered by the lack of program expertise.

The following section mentions some specific recommendations that could be applied to improve the existing generic control system. Some deficiencies in the existing system were revealed during the test implementation of the original generic control system.

Specific Recommendations

Sequence Manager

The existing Sequence Manager has many generic capabilities; however, only digital and analog inputs and outputs are a part of the table driven data structure. Since flexible sequential operation also requires other inputs and outputs, these additional capabilities need to be added to the programmable Sequencer. Inputs from the user (WCC or keyboard), inputs from slave subsystem controllers, flag values, counter values, and additional timer values should be incorporated into the generic structure. Outputs such as, messages and commands to the user (WCC or display), commands to the slave subsystem controllers, and the setting of flags, timers and counters should also be incorporated.

The Sequence Manager should be able to determine the next state based on its input condition and current state in a manner analogous to the finite state machine that it emulates. Presently, next states are only incremented by one. If the next state does not follow the present state, the application-specific code has to be inserted to set the next state of the machine. It is necessary to provide a next state lookup procedure to solve this problem.

Presently, the Sequence Manager does not handle many exception or fault conditions in a generic way. This

problem could be overcome if the mentioned improvements are completed.

Safety Manager

The Safety Manager also has many generic capabilities. It is able to detect system safety fault conditions by reading the System Safety Table and comparing the input data read from the I/O port with the data in the table. The Safety Manager is not able to initiate generic safety routines. For example, if a priority one fault occurs, module outputs are set to a safe state and processing is suspended as it should be. The problem with this system is that the safety routines that are executed as a result of a fault are not a part of the generic Sequence Manager code. These safety shutdown routines could be programmed via the data tables if a more powerful and flexible Sequence Manager were available.

Operator Manager

Each equipment control system will have a different set of displays or messages it sends to its user interface. The system will also have a different set of expected responses from the user. The basic function of the Operator Interface is the same for all equipment; that is, to present the system status to the user and to accept commands to carry out system functions.

The control system designed has some generic Operator Interface functions; however, more flexibility is needed. An ideal Operator Interface would also allow the equipment control system designer to build user screens, for systems requiring a display. These screens would consist of static parts, such as titles, boxes, and labels and dynamic parts, such as real-time data displays. These screen definitions would have to be stored in a file and restored each time the control program was executed. Conversion formulas would also have to be built into the Operator Manager data structures so that dynamically displayed data could be presented in the desired format. The Operator Manager would also have to provide a way for the user to access selected screens. A screen manager data structure with its associated procedures could provide this level of programmablity.

In addition to screen management and screen configuration abilities, a data structure and associated procedures would be required to interpret messages and commands received from the user. These messages would have to be verified before being accepted. If the message or command was invalid, it would have to be ignored and an appropriate response would be sent to the user. The Operator Manager would have to interface with the Sequence Manager since user inputs are required for sequential operation.

Communication Manager

A generic Communication Manager needs to be developed to provide an interface between the equipment and its slave subsystems and between the equipment and the WCC. The Communication Manager would perform the task of converting output messages from the equipment into the correct format for transmission to the receiver. Also, incoming messages would have to be read, buffered, and routed to the desired destination within the control system software. Communication specifics, such as, transmission rate, message format, handshaking, and exception handling would have to be table-driven. Message queue structures would have to be designed to provide temporary storage for messages awaiting transmission and received messages awaiting processing. Due to the variety of communication protocols, the primitive communication routines would probably have to be designed for the particular communication system. The specific drivers needed for a particular application could be referenced via a system configuration table.

Final Remarks

The original objective of this project was to develop a totally generic process control system. A large part of the overall objective has been accomplished. Also, a great amount of experience was obtained in real-time equipment control system design. This experience has provided a

knowledge base from which to specify functional requirements for a much more generic process control system.

Given sufficient resources; such as an experienced programming team, a flexible equipment control system can be developed that would satisfy the control needs for many applications. It is important, however, that industry adopt a general top-down facility plan that specifies the requirements for automatic process equipment in such a way that equipment control designers can meet these demands. Equipment communication standards also need to be well defined and documented to aid in the interconnection of controllable units on the factory floor.

A totally generic equipment control system can be developed and is important to strive for such a system. In the mean time, the attempt to achieve this goal will result in the development of a set of general-purpose tools that can be used to simplify process controller design. The continued effort to develop and refine reusable software tools will simplify the complex process control system designs of the future and provide the needed building blocks for the ultimate goal of a totally generic process control system.

LIST OF REFERENCES

[1] Deninger, R. "Specifying a Process Automation System." Microelectronics Manufacturing and Testing, Jan. 1988, pp 88-97.

[2] Hughes, R. "The Future of Automation for High-Volume Wafer Fabrication and ASIC Manufacturing." Proceedings of the IEEE, Vol. 74, No. 12, Dec. 1986, pp 1775-1793.

[3] Burggraff, P. "Semiconductor Factory Automation: Current Theories." Semiconductor International, Oct. 1985 pp 88-97.

[4] Arrant, E. "Generic Factory Control." Proceedings of SEMI Automation Forum 89 - The Gateway to the 90's, April 1989.

[5] Morris, H. "Profitable Robotic Work Cells Result From Interconnecting the Islands of Automation." Control Engineering, May 1985, pp 81-84.

[6] Pelusi, J. "Cell Control - The High and Low of It." Chiltons I & CS Control Technology for Engineers and Engineering Management, Aug. 1989, pp 37-38.

[7] Babb, M. "Riding the PC Bus: New Avenues to Industrial Automation." Control Engineering, Oct. 1986, pp 64-65.

[8] Arrant, E. "An Automatic Control System for Semiconductor Processing Equipment." Proceedings of SEMICON Southwest, 1986, Oct. 1986.

[9] Shoch, J. "Evolution of the Ethernet Local Computer Network." Computer, Aug 1982, pp 10-27.

[10] Richardson, D. "Implementing MAP for Factory Control." Manufacturing Engineering, Jan. 1988, pp 79-82.

[11] Parten, M. "MAP TOP and SECSII." Proceedings of SEMICON Southwest, 1987, Oct. 1987, pp 201-208.

[12] Fletcher, W. An Engineering Approach to Digital Design Prentice-Hall, Inc., 1980, pp 208,209,293,338.

[13] Chang, H. "A Time Shared Automatic Controller for Multiple Module Semiconductor Processing." Proceedings of SEMICON Southwest, 1987, Oct. 1987, pp 75-81.

[14] Koivo, H. "Microcomputer Real-Time Multitasking Operating System." Computers in Industry, 1985, pp 32-33.

[15] "Etching of SiO2 by Controlled Anhydrous HF/Vapor." Semiconductor International, Sept. 1987, pp 80.

[16] "FSI Excalibur Vapor Etching System." Microcontamination, Oct. 1987, pp 50.

[17] FSI International Office Memo, Chaska, MN, 1989.

[18] Moser, P. "Computer Controlled Plasma Reactor System Proposal." Texas Tech University, Aug. 1988, pp 1-31.

[19] Pan, Y. "Monitoring and Control of Semiconductor Manufacturing Coating and Developing Process." Master's Thesis in Electrical Engineering, Texas Tech University, Aug. 1988, pp 72-96.

[20] FSI International (Advanced Technology Center) Office Memo, Lubbock, TX, 1989.

[21] Parten, M. "Multi-tasking Automatic Control for a Semiconductor Chemical Processing System." High Technology Computer Conference - Automation '86, March 1986, pp 310-314.

[22] The Multi-Tasking and Real-Time Library Users Guide, TimeSlicer, Inc., 1986, pp 5-14.

[23] Data Acquisition and Industrial Control Interfaces for IBM PC/XT/AT, Compatibles, & Apple II. MetraByte Corporation, Spring 1986 Catalogue, pp 80-109.

APPENDIX

PARTIAL PROGRAM LISTING

Header File Digital.H

```
#ifdef MAIN
/* File                 struct.h = extn.h
 * Revision date        9-5-86, 3-24-88
 * Original date        3-15-86
 * Purpose              to define "digital" structures
 *                          1) SOFTWARE MANAGER1  ( swm10 )
 *                          2) PROCESS MANAGER    ( pm_sli )
 *
 * Revision date :      2-20-87
 * Purpose       :      for use in checker() function
 *
 * Revision date :      4-16-87
 * Purpose       :      Move in_data & out_data struct into
 *                      struct st0  for logical needs
 * Revision  date:      4/30/87
 * Purpose       :          NUM_ACT=5  to 8 NUM_S=10 to 20,
 *                          NUM_F=8 to 16
 *                          NAMESIZE=20 to 16
 *
 */

int p,f,s,a,xx;

int NUM_ACT=8, NUM_S=20, NUM_F=16, NUM_P=3, NAMESIZE=16;

struct st2 { int n_dis, n_dos;
             int i_p2[8], set_vdi[8];
             int o_p2[8], set_vdo[8];
           } ;

struct st1 { int uf_n;
           };

struct out_data
           { char o_name[16];
             int o_addr, o_mask;
             int t_d, t_out;
           } ;

struct in_data
           { char in_name[16];
             int in_addr, in_mask;
             int f_code;
           } ;
```

```
struct st0 { struct st1 F[16];
             char p_name[16];
             int n_function;
             int io_p1;
             struct in_data in[24];
             struct out_data out[24];
           } ;

struct uni_fn
           { struct st2 S[20];
             char uf_name[16];
             int n_step, n_dif, n_dof;
           } ;

struct sys_input
     { int s_incode,s_tdelay,s_fcode,s_fvalue,s_fdf,s_fpf;
          long s_tdtic;
       int sys_da_flag;
       int sa_flt_hi;
       int sa_flt_low;
     } ;

struct sysin_d { struct sys_input sysin[20]; };
struct sysin_d *dsin;

struct hw_in                          /* HW inadr pter str */
    { int in_ptr[12][20];  };  /* 0-11 funs, 0-19 ptr/fn */
struct hw_in *hw_in0;                 /* ptr to hw_in struct */

struct hw_out
    { int out_ptr[12][20]; };
struct hw_out *hw_out0;

struct hf_bottle
    { char inst_date[10];
      float inst_weight;
      float dplt_limit;
    } ;

/* int i_cd[50][20];  */
/* int o_cd[50][20];   */
struct st0 *pt0;
struct uni_fn *uf;
struct hf_bottle *hf0;
```

```
#else

/* File                 struct.h
 * Revision date        9-5-86, 3-24-88
 * Original date        3-15-86
 * Purpose              to define "digital" struct & MACROS
 *                          1) SOFTWARE MANAGER1  ( swm10 )
 *                          2) PROCESS MANAGER    ( pm_sli )
 *
 * Revision date :      2-20-87
 * Purpose       :      for use in checker() function
 * Revision  date:      4/30/87
 * Purpose       :      Change  NUM_S=10 to 20,NUM_F=8 to 16
 *                               NAMESIZE=20 to 16
 *
 */

extern int p,f,s,a,xx;

extern int NUM_ACT, NUM_S, NUM_F, NUM_P, NAMESIZE;

extern struct st2 { int n_dis, n_dos;
              int i_p2[8], set_vdi[8];
              int o_p2[8], set_vdo[8];
            } ;

extern struct st1 { int uf_n;
                  };

extern struct out_data
            { char o_name[16];
              int o_addr, o_mask;
              int t_d, t_out;
            } ;

extern struct in_data
            { char in_name[16];
              int in_addr, in_mask;
              int f_code;
            } ;

extern struct st0 { struct st1 F[16];
              char p_name[16];
              int n_function;
              int io_p1;
              struct in_data in[24];
              struct out_data out[24];
            } ;
```

```
extern struct uni_fn
           { struct st2 S[20];
             char uf_name[16];
             int n_step, n_dif, n_dof;
           } ;

extern struct sys_input
      { int s_incode,s_tdelay,s_fcode,s_fvalue,s_fdf,s_fpf;
            long s_tdtic;
        int sys_da_flag;
        int sa_flt_hi;
        int sa_flt_low;
      } ;

extern struct sysin_d { struct sys_input sysin[20]; };
extern struct sysin_d *dsin;

extern struct hw_in              /* HW inadr pointer str */
    { int in_ptr[12][20];  }; /* 0-11 fun, 0-19 ptr/fn */
extern struct hw_in *hw_in0;  /* ptr to hw_in struct */

extern struct hw_out
    { int out_ptr[12][20]; };
extern struct hw_out *hw_out0;

extern struct hf_bottle
    { char inst_date[10];
      float  inst_weight;
      float  dplt_limit;
    };

/* extern int i_cd[][20]; */
/* extern int o_cd[][20];  */

extern struct st0 *pt0;
extern struct uni_fn *uf;
extern struct hf_bottle *hf0;

#endif

#define P_NAME(p)        (pt0+p)->p_name
#define N_F(p)           (pt0+p)->n_function

#define UF_N(p,f)        (pt0+p)->F[f].uf_n
#define IO_P1(p,f)       (pt0+p)->io_p1+f
#define UF_NAME(p,f)     (uf+((pt0+p)->F[f].uf_n))->uf_name
#define N_STEP(p,f)      (uf+((pt0+p)->F[f].uf_n))->n_step
#define N_DIF(p,f)       (uf+((pt0+p)->F[f].uf_n))->n_dif
#define N_DOF(p,f)       (uf+((pt0+p)->F[f].uf_n))->n_dof

#define N_DIS(p,f,s)   (uf+((pt0+p)->F[f].uf_n))->S[s].n_dis
```

```
#define N_DOS(p,f,s)  (uf+((pt0+p)->F[f].uf_n))->S[s].n_dos

#define I_P2(p,f,s,a)
            (uf+((pt0+p)->F[f].uf_n))->S[s].i_p2[a]
#define O_P2(p,f,s,a)
            (uf+((pt0+p)->F[f].uf_n))->S[s].o_p2[a]
#define SET_VDI(p,f,s,a)
            (uf+((pt0+p)->F[f].uf_n))->S[s].set_vdi[a]
#define SET_VDO(p,f,s,a)
            (uf+((pt0+p)->F[f].uf_n))->S[s].set_vdo[a]

#define DUMIN(p,f,s,a)
             (hw_in0+p)->in_ptr[f][I_P2(p,f,s,a)]
#define DUMOUT(p,f,s,a)
             (hw_out0+p)->out_ptr[f][O_P2(p,f,s,a)]

#define IN_NAME(p,f,s,a)
                (pt0+p)->in[DUMIN(p,f,s,a)].in_name
#define IN_ADDR(p,f,s,a)
                (pt0+p)->in[DUMIN(p,f,s,a)].in_addr
#define IN_MASK(p,f,s,a)
                (pt0+p)->in[DUMIN(p,f,s,a)].in_mask
#define F_CODE(p,f,s,a)
                (pt0+p)->in[DUMIN(p,f,s,a)].f_code
#define O_NAME(p,f,s,a)
                (pt0+p)->out[DUMOUT(p,f,s,a)].o_name
#define O_ADDR(p,f,s,a)
                (pt0+p)->out[DUMOUT(p,f,s,a)].o_addr
#define O_MASK(p,f,s,a)
                (pt0+p)->out[DUMOUT(p,f,s,a)].o_mask
#define T_OUT(p,f,s,a)
                (pt0+p)->out[DUMOUT(p,f,s,a)].t_out
#define T_D(p,f,s,a)
                (pt0+p)->out[DUMOUT(p,f,s,a)].t_d

#define S_INNAME(p,xx)
   (pt0+p)->in[(dsin+p)->sysin[xx].s_incode].in_name
#define S_INADR(p,xx)
   (pt0+p)->in[(dsin+p)->sysin[xx].s_incode].in_addr
#define S_INMASK(p,xx)
   (pt0+p)->in[(dsin+p)->sysin[xx].s_incode].in_mask

#define S_INCODE(p,xx)  (dsin+p)->sysin[xx].s_incode
#define S_TDELAY(p,xx)  (dsin+p)->sysin[xx].s_tdelay
#define S_FCODE(p,xx)   (dsin+p)->sysin[xx].s_fcode
#define S_FVALUE(p,xx)  (dsin+p)->sysin[xx].s_fvalue
#define S_FDF(p,xx)     (dsin+p)->sysin[xx].s_fdf
#define S_FPF(p,xx)     (dsin+p)->sysin[xx].s_fpf
#define S_TDTIC(p,xx)   (dsin+p)->sysin[xx].s_tdtic
```

```
#define SA_NAME(p,xx)
   ain[p][(dsin+p)->sysin[xx].s_incode].a_name
#define SA_INADR(p,xx)
   ain[p][(dsin+p)->sysin[xx].s_incode].ain_addr
#define SA_INCHAN(p,xx)
   ain[p][(dsin+p)->sysin[xx].s_incode].ain_chan
#define SA_F_HI(p,xx)
   (dsin+p)->sysin[xx].sa_flt_hi
#define SA_F_LO(p,xx)
   (dsin+p)->sysin[xx].sa_flt_low
#define SA_DA_FLAG(p,xx)
   (dsin+p)->sysin[xx].sys_da_flag

/* hf bottle struct defines */
#define HF_IN(p)              (hf0+p)->inst_date
#define HF_WT(p)              (hf0+p)->inst_weight
#define HF_LT(p)              (hf0+p)->dplt_limit
```

Header File Analog.h

```
#ifdef MAIN
/* struct_a.c                                                    */
/* 8-7-86    (by EKA & Xu)                                       */
/* 4-86      (by Xu)                                             */
/* to define "analog" structures and MACROS  for                 */
/*       1) SOFTWARE MANAGER2  (swm20)                           */
/*       2) PROCESS MANAGER    (pm_sli)                          */

int etchds,euf_code,efn;

struct adata_set   {  int exec_t;
                      int setup_tm;
                      int set_vai[6];
                      int set_vao[6];
                   };

/* 10 recipes per sequence */
struct ast1        {  struct adata_set aset[10];
                      int n_ais, n_aos;
                      int aic[6];
                      int aoc[6];
                   };

struct ast2    {  struct ast1 as[20];  /* 20 steps/rec */
               };

char rcp_name[100][16];

struct ain_data    {  char a_name[16];
                      int ain_addr;
                      int ain_chan;
                      int af_code;
                   }  ain[4][20];

struct aout_data   {  int aout_addr;
                      int aout_bd;
                      int setup_t;
                   }  aout[4][20];

struct ast2 *af_st;
```

```
#else

/* File                 struct_a.c =extn1.h
 * Revision date        8-7-86   (by EKA & Xu)
 * Original date        4-86     (by Xu)
 * Purpose              to define "analog" struct & MACROS
 *                           1) SOFTWARE MANAGER2  (swm20)
 *                           2) PROCESS MANAGER    (pm_sli)
 * Rev date: 4/23/87
 */

extern int etchds,euf_code,efn;

extern struct adata_set   {  int exec_t;
                             int setup_tm;
                             int set_vai[6];
                             int set_vao[6];
                          };

extern struct ast1        {  struct adata_set aset[10];
                             int n_ais, n_aos;
                             int aic[6];
                             int aoc[6];
                          };

extern struct ast2        {  struct ast1 as[20];
                          };

extern char rcp_name[100][16];

extern struct ain_data    {  char a_name[16];
                             int ain_addr;
                             int ain_chan;
                             int af_code;
                          }  ain[4][20];

extern struct aout_data   {  int aout_addr;
                             int aout_bd;
                             int setup_t;
                          }  aout[4][20];

extern struct ast2 *af_st;

#endif

#define N_AIS(efn,s)        (af_st+efn)->as[s].n_ais
#define N_AOS(efn,s)        (af_st+efn)->as[s].n_aos
#define AIC(efn,s,a)        (af_st+efn)->as[s].aic[a]
#define AOC(efn,s,a)        (af_st+efn)->as[s].aoc[a]
```

```
#define EXE_T(efn,s,etchds)
   (af_st+efn)->as[s].aset[etchds].exec_t
#define SETUP_TM(efn,s,etchds)
   (af_st+efn)->as[s].aset[etchds].setup_tm
#define SET_VAI(efn,s,a,etchds)
   (af_st+efn)->as[s].aset[etchds].set_vai[a]
#define SET_VAO(efn,s,a,etchds)
   (af_st+efn)->as[s].aset[etchds].set_vao[a]
#define AI_NAME(mn,efn,s,a) ain[mn][AIC(efn,s,a)].a_name
#define AF_CODE(mn,efn,s,a) ain[mn][AIC(efn,s,a)].af_code
#define AI_ADDR(mn,efn,s,a) ain[mn][AIC(efn,s,a)].ain_addr
#define AI_CHAN(mn,efn,s,a) ain[mn][AIC(efn,s,a)].ain_chan
#define AO_NAME(mn,efn,s,a) ain[mn][AOC(efn,s,a)].a_name
#define AO_ADDR(mn,efn,s,a)
                aout[mn][AOC(efn,s,a)].aout_addr
#define AO_BD(mn,efn,s,a)   aout[mn][AOC(efn,s,a)].aout_bd
#define SETUP_T(mn,efn,s,a) aout[mn][AOC(efn,s,a)].setup_t
```

```
/* MAIN Program of Generic Control */


main()
{ void opm(), pmgr(), smtsk();
  char *calloc();
  char c;
  int i,j;

     initialize();
     load_data_struct();           /* load data files  */
     set_up_com();                 /* setup comm */
     del(1,80,1,25);
     spawn("smtsk",0x800,smtsk);
     mod_select();                 /* select active mod */
     init_qtable();                /* initializes q table */
     set_init_cond();              /* initial conditions */
     ss=0;last_sec=0;
     spawn("pmgr",0x2000,pmgr);  /* spawn process mger */
     spawn("opm",0x1000,opm);    /* spawn operator mger */
}
```

```
void pmgr()
{
  do{ getime(&t);
      for(ql=0;ql<4;ql++)              /* scan qt lines */
      { if (bp_control[ql]==1)         /* PID control   */
         { if (old_ts[ql] !=ts)        /* Resolution tick */
           { old_ts[ql]=ts;            /* perform PID con */
             do_flow_PID(ql);
           }
         }
       chk_mod_inputs();              /* check module ins */
       exec_pqt();                    /* execute proc qt  */
       if(safety_flag[ql]!=0) safety();  /* safety */
    }
    guc();                             /* give up control */
  } while (1);
}
```

```
/*.
step_exec()
Step_exec() or Step Execution is a table
 driven digital input and output routine.  Step_exec()
 reads a group of inputs from the Function Data Table
 which are referenced for the current step number
 If the inputs that are read are TRUE, then the outputs
 referenced for that step number will be set to '1' or
 '0' depending on the data in the Function Data Table .
 If the inputs that are read are FALSE, then step_exec()
 will check if a timeout has occured.  A timeout is the
 max. time allowed for a certain input to become VALID or
 for that input to read equal to the (programmed) desired
 value.  If the timeout has expired, then a fault conditio
 exists and a fault set routine (digital_f_set() ) is
 called, which results in the suspension of exec_pqt()
 until prior2() ( sequence safety routine ) has completed.
 (get list; link-link-link; soca-doca)
*/

void step_exec()
{
  input_verify();                    /* chks inputs */
  if (in_error !=0 )                 /* any errors ? */
  { if (ptime>qt[ql].tout)           /* time out expired ? */
    {
     digital_f_set();                  /*  set digital flt info */
     return;
    }
    else return ;                   /* time out NOT expired  */
                                    /* and chk again later   */
  }

  /* Jumps to here if no input errors  */

  if (N_DOS(pq,fq,sq) !=0)    /* any digital outputs ?*/
  { set_dos();                /* set digital outs */
                            /* load qtable w/ tdelay/tout */

    qt[ql].tout=ptime+T_OUT(pq,fq,sq,0);
    qt[ql].tdelay=ptime + T_D(pq,fq,sq,0);
  }
  qt[ql].sn++ ;                          /* increment step # */
}
```

```
/*.
void input_verify()

Input_verify() is a table driven digital input routine
which reads and verifies the digital inputs for a given
function (ex module_load() ) and step number (qt[ql].sn).
The qtable contains the info. needed to look up the step
input data.  If any input is read to be NOT equal to the
desired value, 'in_error' is set to ONE which flags the
calling routine that an error exists.  The calling routine
decides what action to take based on the value returned
thru the global variable, 'in_error'.
*/

void input_verify()                    /* verify inputs okay */
{
  int bitin,xin,des_mb_adr;
  pq=qt[ql].pn;
  fq=qt[ql].fn;sq=qt[ql].sn;
  in_error=0;                          /* error counter */
  if (N_DIS(pq,fq,sq)==0) return;      /* digital inputs ?  */
                                       /* loop to read inps */
  for (a=0;a<N_DIS(pq,fq,sq);a++)
  {
    des_mb_adr=IN_ADDR(pq,fq,sq,a);    /* set mb adr */
    xin = inmb(des_mb_adr);            /* get byte from mb*/
    xin=xin & IN_MASK(pq,fq,sq,a);     /* bit read mask   */
    if(xin>0) bitin=1;                 /* actual bit value*/
    else    bitin=0;
    if(bitin != SET_VDI(pq,fq,sq,a))   /* != des val ? */
    { in_error++;                      /* set error flag */
      return;                          /* 1st flt detected*/
    }
  }
}

/*.
void set_dos()
Set Digital Outputs or set_dos() is the partner routine
to input_verify().  Set_dos() sets the digital outputs
referenced in the Function Data Table.  The specific
location in the Function Data Table is referenced
by the function and step numbers which are stored in
the qtable.
To set a digital output bit, it is necessary to know
the byte and bit address of the bit of interest.
To write to the Metrabus, the entire byte must be
sent; therefore, it is important that the other bits
NOT be changed.
*/
```

```
void set_dos()
{
  int out_addr,xout,des_mb_adr;
  if(N_DOS(pq,fq,sq)==0) return;    /* # dig outs = 0 ? */
  for (a=0;a<N_DOS(pq,fq,sq);a++)   /* set digital outs */
  {
    /* desired output val=1 ? */
    /*  then outbyte=present port val ORed w/ bit mask */
    /*  else outbyte=present port val ANDed w/ mask comp */

    out_addr=O_ADDR(pq,fq,sq,a);     /* out address */
    if (SET_VDO(pq,fq,sq,a)==1)      /* out bit = 1 ? */
     xout=port_data[out_addr] |      /* then 'or' w/ mask */
          O_MASK(pq,fq,sq,a);
    else                             /* 'and' w/ mask comp */
     xout=port_data[out_addr] & (255 -
          O_MASK(pq,fq,sq,a));
    /*  send out byte to port  */
     des_mb_adr=out_addr;            /* mb addr */
     outmb(des_mb_adr,xout);         /* send byte to mb */
     port_data[out_addr] = xout;     /* rev. portdata */

  }
}
```

```
/*.
void a_step_exec()
The routine, Analog Step Execution ( a_step_exec() ),
is the main part of etch().  This routine checks
digital inputs, looking for chamber closed and not
open, and sets a fault if a one exists.  These digital
inputs are checked with input_verify().
The digital and analog outputs are set according to
The data in the Function Data Table and the (Analog)
Recipe Data Table.  The step end time is calculated
by adding the present time and the step with the result
stored in the qtable for future reference.  Once the
step time expires ( step end time = present time ),
the step ends and the next step will begin.
Analog inputs are checked after a 'set-up-time'
which allow the MFCs to reach set point before
verifing their setpoint value(s).  If a MFC fails
to reach setpoint, prior4print() is executed, which
prints the MFC fault info. to the operator screen. The
step number is then set to the purge step. A flag is
set (qt[ql].wnef or qt[ql].wdf) which signals a MFC
fault recovery routine prior4() of the nature of the
MFC fault. Prior4() safety routine will be turned on
after the wafer has been returned to the carrier.
```

```
This routine is a two-entry point routine, which
makes use of a 'first pass flag' (qt[ql].f_p_flag).
The first time the routine is called (for each step)
the 1st pass flag=1, which will cause the digital and
anaglog outputs to be set and the step end time to
be saved.  The proceeding pass thru the routine,
which occur once per sec, will skip the 1st pass section
because the 1st pass flag has been set to 0.  The
remainning passes thru a_step_exec() are to verify
that the chamber remains closed ,that the MFCs are
at set point, and to read/display the B.P. and MFC values.
*/

void a_step_exec()              /*  analog step execution  */
{ int ret,bp_xin;
  efnq=qt[ql].ef_no;            /* etch fn  # this QT line  */
  dq=qt[ql].ds_no;              /* data set # this QT line  */
  pq=qt[ql].pn;                 /* process  # "           "  */
  fq=qt[ql].fn;                 /* function # "           "  */

  sq=qt[ql].sn;                 /* step #  */

  input_verify();               /*  check digital inputs    */
  if (in_error !=0)             /*  any digital in errors ? */
{ if(F_CODE(pq,fq,sq,a)==1)
  { digital_f_set();     /* fcode=1 ?  (cham fault) */
  }
  else
  { writemsg(ql,12,2,IN_NAME(pq,fq,sq,a),TEXT2);
    timestr();
    writemsg(ql,12,19,timech,TEXT2);
    etch_active();
  }
  return;
}

bp_xin=bp_read();             /* read/display back pres */

if (qt[ql].f_p_flag==1)       /* step's first pass ? */
{
  qt[ql].f_p_flag=0;          /*  clear first pass flag  */
  set_dos();                  /*  set digital outputs  */
  set_aos();                  /*  set analog outputs   */
  qt[ql].s_e_tm=ptime         /*  step end time to QT  */
   + EXE_T(efnq,sq,dq);
  qt[ql].s_u_tm=ptime         /*  step setup time to QT */
   + SETUP_TM(efnq,sq,dq);
 }
```

```
    a_in_verify();              /* check analog inputs  */
    if (in_error !=0   )        /*  if errors  */
    { if((ret=prior4print())==1)
      return;                   /*  end time */
    }
    if (bp_control[ql]==1)
    { chk_bp_ok(bp_xin);          /* chk bp ok*/
      if(bp_fault[ql]==1)         /* first det BP fault */
      { if((ret=prior4print())==1)   /*  prt msg  */
        return;
      }                          /* ignore BP faults */
    }
    if (ptime >= qt[ql].s_e_tm)  /* step time expired ? */
    { end_a_step();              /* end step & chk BP cond*/
      return;
    }
    if (ptime + re_td > qt[ql].s_e_tm)
        qt[ql].tdelay=qt[ql].s_e_tm;    /* ret at step end*/
    else
       qt[ql].tdelay=ptime + re_td;    /* return in 1 sec */
    return;
}
```

```
/*.
void set_aos()
Set Analog Outputs (set_aos()) is a table driven
routine that sets analog output values for the
etch() routine.  The MFC setpoint data is obtained
from the Recipe table, which is referenced by
recipe number and step number.  The set point data
is displayed to the operator and monitor screens.
*/

void set_aos()                        /* set analog outputs */
{ char ver[8];
  int iver,xout,des_mb_adr,a1,i;

  if (N_AOS(efnq,sq)==0) return;       /* # analog out=0  */
  for(a1=0;a1<N_AOS(efnq,sq);a1++)     /* set ana outs */
  { xout=SET_VAO(efnq,sq,a1,dq);       /* send on mb */
    des_mb_adr=AO_ADDR(ql,efnq,sq,a1);   /* set mb addr  */
    outmb(des_mb_adr,xout);              /* send byte    */
    port_data[AO_ADDR(ql,efnq,sq,a1)]=xout;   /* save    */
    switch(AO_ADDR(ql,efnq,sq,a1))            /* select  */
    { case 0x26:case 0x22:case 0x2e:case 0x2a:
           i=0;break;                              /*  n2   */
      case 0x27:case 0x23:case 0x2f:case 0x2b:
           i=1;break;                              /* vap   */
```

```
    case 0x24:case 0x20:case 0x2c:case 0x28:   /* HF1  */
         i=2;
         if(xout==0) sethf(hf1_outcode[ql],0);
         else        sethf(hf1_outcode[ql],1);
         break;
    case 0x25:case 0x21:case 0x2d:case 0x29:   /* HF2  */
         i=3;
         if(xout==0) sethf(hf2_outcode[ql],0);
         else        sethf(hf2_outcode[ql],1);
         break;
  }
  iver= (int)(xout/2.55+0.5);
  sitoa(iver,ver);
  deletemsg(ql,37,48,15+i,15+i);    /* erase old value */
  writemsg(ql,15+i,37,ver,TEXT2);   /* print mfc data */
  strcpy(mfc_array[ql][0][i],ver);  /* prt mfc to monit*/
}
i=4;
iver=(int)((qt[ql].bp-qt[ql].bp_offset)/6.144+0.5);
sitoa(iver,ver);
deletemsg(ql,37,41,15+i,15+i);     /* erase bp value */
if (bp_control[ql]==1)
{  writemsg(ql,15+i,37,ver,TEXT2);  /* print set data */
   strcpy(mfc_array[ql][0][i],ver); /* bp outs to monit*/
  }
  else
  { writemsg(ql,15+i,37,"   ",TEXT2); /* prt bp set data */
    strcpy(mfc_array[ql][0][i],"  "); /* bp outs to monit*/
  }
}

/*.
p2.06            void a_in_verify()
Analog Input Verify (a_in_verify()) check the analog
input values and compares these values with the
set point values.  If the input values are NOT with in
+ or - 2% of full scale, a MFC fault is said to exist.
*/

void a_in_verify()              /* verify analog inputs */
{ int iver,xin,des_mb_adr,ainchannel,a1,i;
  char ver[8];
  in_error=0;                          /* clr error counter  */
  if (N_AIS(efnq,sq)==0)  return; /* # analog ins = 0 ? */

  for(a1=0;a1<N_AIS(efnq,sq);a1++)       /* get ana.in's */
  { des_mb_adr=AI_ADDR(ql,efnq,sq,a1);  /* set mb addr */
    ainchannel=AI_CHAN(ql,efnq,sq,a1);  /* ana.in channel*/
    xin=getain(des_mb_adr,ainchannel);  /* analog in data*/
    setvai=SET_VAI(efnq,sq,a1,dq);      /* analog data */
    iver=(int)((xin -2048)/10.24+0.5);
```

```
sitoa(iver,ver);

switch(ainchannel)    /* select prt-line pointer */
{ case 10: case 2: i=0;break;                  /*  n2   */
  case 11: case 3: i=1;break;                  /* vap   */
  case  8: case 0: i=2;break;                  /* hf 1 */
  case  9: case 1: i=3;break;                  /* hf2  */
}
deletemsg(ql,45,48,15+i,15+i);
writemsg(ql,15+i,45,ver,TEXT2);          /* prt rates */
strcpy(mfc_array[ql][1][i],ver);
if(ptime<qt[ql].s_u_tm ||                /* no chk ? */
         qt[ql].wnef==1) continue;
if(setvai==2048 &&
     (xin-2048) < 13*a_er_val) continue;
if(a1==hf1code[ql] &&
   (hf1_2[qt[ql].ef_no]!=1 &&
    hf1_2[qt[ql].ef_no]!=3 ) ) continue;
if(a1==hf2code[ql] &&
   (hf1_2[qt[ql].ef_no]!=2 &&
    hf1_2[qt[ql].ef_no]!=3 ) ) continue;
if((xin-setvai)>a_er_val ||
     (setvai-xin) > a_er_val )
{ in_error++;
      fltname[ql]=AI_NAME(ql,efnq,sq,a1);
      p4desval[ql]=(int)((setvai-2048)/10.24+0.5);
      p4actval[ql]=(int)((  xin -2048)/10.24+0.5);
      if(AF_CODE(ql,efnq,sq,a1)==5 &&
         sq>=hfpurges[qt[ql].ef_no] && p4actval[ql]<18 )
           pff[ql]=1;
      return;
    }
  }
}

/*.
void chk_mod_inputs()
Check module inputs is called by pmgr() to check the
system or module inputs for each module.  Module
inputs such as Air Pressure, N2 Pressure, Hot Plate Temp,
and Water Level are checked each time the Process Manager
services the module.  If a fault is detected, and remains
faulted for the 'timedelay' period, chk_mod_inputs()
considers that input to be faulted.  Both analog and
digital system(module) inputs are checked.

If the faulted inputs has a priority 1 then prior1()
is enabled ( from fsub1 ) and if the faulted input
has a priority 3 then prior3print is called to print
the fault.  Prior3() will not be enable until
a new carrier is loaded. */
```

```
void chk_mod_inputs()              /* CHECK MODULE INPUTS */
{ int xin,bitin;

  if(qt[ql].pn==-1)return;                    /* inactive ?  */
  for(slc=0;slc<num_s_inputs[ql];slc++)       /* scan sys in */
  { if(SA_DA_FLAG(ql,slc)==0)                 /* digital?*/
    { xin=inmb(S_INADR(ql,slc) )  &           /* sys input byte*/
        S_INMASK(ql,slc);                     /* &  with mask  */
    if(xin>0) bitin=1;                        /* input bit = 1 */
    else      bitin=0;                        /*   or zero ?   */
    if(bitin==S_FVALUE(ql,slc) &&             /* input = fault */
       ignore_s_flt[ql][slc]==0   )           /* ignore NOT set*/
    { fsub1();                                /* then flt sub 1*/
      continue;
    }
    S_FPF(ql,slc)=0;                    /*  clr first pass */
  }
  else                                   /* analog input chk */
  { xin=getain(SA_INADR(ql,slc),
               SA_INCHAN(ql,slc) ); /* read analog value*/
    if(xin>SA_F_HI(ql,slc)    ||
       xin<SA_F_LO(ql,slc)       )  /* input out of spec */
    {
      fsub1();                          /*  then fault sub 1 */
      continue;
    }
    S_FPF(ql,slc)=0;                    /* clr first pass */
  }
 }
}

/*.
void  fsub1()
Fsub1() is called by chk_mod_inputs() in case of a faulted
module input
*/

void fsub1()     /* fault sub 1 (digital) */
{
  if(S_FDF(ql,slc)==1 ||          /* flt det flg set ? */
   S_FCODE(ql,slc)==0 )  return;   /* or flt code==0  */
  if(S_FPF(ql,slc)==0)             /* first pass set ? */
  {  S_FPF(ql,slc)=1;              /*  then set it */
     S_TDTIC(ql,slc)=
     ptime+S_TDELAY(ql,slc);       /* set sys fault delay*/
  }
  if(S_TDTIC(ql,slc)<=ptime)       /* sys tdelay expired?*/
  { S_FDF(ql,slc)=1;               /* sys fault detect */
```

```
    switch(S_FCODE(ql,slc))          /* select flt routine*/
    {
    case 1:                          /* Prior1 */
     if(slc==epo_slc)
     { setportdata_0(ql);            /* then clr port data */
       qt[ql].f_flag=1;              /*   set seq flt flag*/
       p6_fcode[ql]=1;               /*   p6=home handler */
       safety_flag[ql]=
        safety_flag[ql] | 0x20;      /* set p6 bit is safety*/
     }
     if(qt[ql].mff==0)               /* mff not set yet ? */
     { qt[ql].mff=1;                 /*  then set mff */
       p1_f_slc[ql]=slc;             /*  tag flted sys in */
                                     /*   (enables prior1)*/
       safety_flag[ql]=
         safety_flag[ql] | 0x01;     /* prior1 bit in flag */
     }
     break;
    case 3:                          /* Proir3  */
     prior3print;                    /* print p3 fault msg */
     break;                          /*  and set BFF */
  }
 }
}
```

```
/*.
void safety()
Safety() is called by pmgr(), (Process Manager) if the
safety_flag[] is NONZERO.  If safety() is called, then
any safety routine (prior0()-prior12()) will be called
if that prior*() has it's p*_fcode[] set. Each prior*()
routine is a seperate safety routine which may or may
not be active.  If all safety routine are not active,
then safety_flag[] = 0 and safety will not be called
and the CPU does not have to check all 12 routines
to see if any are active.  Each prior*() routine
has assigned to it a unique bit in safety_flag[] which
allows any prior routine the ability to turn on safety
so that the enabled prior routine can be called.
*/

void safety()   /* SAFETY ROUTINE SELECT */
{
 if(safety_flag[ql]!=0x400) flash();

 if(p0_fcode[ql]!=-1)  prior0();
 if(p1_f_slc[ql]!=-1)  prior1();   /* prior1 if p1 set */
 if(p2_fcode[ql]!=-1)  prior2();   /* prior2 if p2 set */
 if(p3_fcode[ql]!=-1)  prior3();   /* prior3 if p3 set */
 if(p4_fcode[ql]!=-1)  prior4();   /* prior4 if p4 set */
 if(p6_fcode[ql]!=-1)  prior6();   /* prior6 if p6 set */
```

```
 if(p7_fcode[ql]!=-1)  prior7();   /* prior7 if p7 set */
 if(p8_fcode[ql]!=-1)  prior8();   /* prior8 if p8 set */
 if(p9_fcode[ql]!=-1)  prior9();   /* prior9 if p9 set */
 if(p10_fcode[ql]!=-1) prior10();  /* prior10 if p10 set */
 if(p11_fcode[ql]!=-1) prior11();  /* prior11 if p11 set */
 if(p12_fcode[ql]!=-1) prior12();  /* prior12 if p12 set */
}
```

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Disagree (Permission not granted)	Agree (Permission granted)
_____	Elsie K Grant
Student's signature	Student's signature
_____	_____
Date	Date