1989

An expert systems based automatic control software generator for the Programmable Logic Controller

Sangeet Bhatnagar
Iowa State University

An expert systems based automatic control software generator for the Programmable Logic Controller

by

Sangeeta Bhatnagar

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Major: Industrial Engineering

Iowa State University
Ames, Iowa
1989

TABLE OF CONTENTS

1 INTRODUCTION 1

2 REVIEW OF RELEVANT LITERATURE 4

2.1 Historical Background 4

2.2 Expert Systems in Industrial Control 10

2.3 Automatic Code Generators 12

3 METHODOLOGY 18

3.1 General Structure of the Software Generator 18

3.1.1 Introduction 18

3.1.2 The user interface module 23

3.1.3 The wiring instruction 33

3.1.4 The error checking module 34

3.1.5 The task code module 35

3.2 Implementation 35

3.2.1 The user interface module 35

3.2.2 The error checking module 39

3.2.3 The task code module 41

3.2.4 Some general comments on the user interface module and task code module designs 45

4 RESULTS 50

4.1 A Complete Example 50

4.1.1 Explanation of the process 51

4.1.2 Process description used by the user interface module 53

4.1.3 Intermediate functional code 55

4.1.4 Wiring instruction 56

4.1.5 Task codes 56

4.1.6 Ladder Logic Diagram 57

5 CONCLUSION 59

5.1 Future Research 60

6 APPENDIX A USER MANUAL FOR THE USER INTERFACE MODULE 62

7 APPENDIX B USER MANUAL FOR THE TI-530 PLC . . . 71

8 APPENDIX C TASK CODES FOR THE TI-530 PLC 80

9 BIBLIOGRAPHY 87

LIST OF TABLES

Table 3.1: Binary input and output devices 26

Table 3.2: Keywords and their attributes 47

Table 3.3: Operation description format 48

Table 3.4: Device, instruction, and function symbols 49

Table 4.1: System specification 53

Table 4.2: Wiring instruction for example system specification 56

Table 6.1: Keywords and their attributes used in the system description 64

Table 6.2: Operation description format 65

Table 6.3: Common PLC internal devices and instructions 65

Table 6.4: Single input functions and their operands 66

Table 6.5: Mutiple input functions and their operands 70

Table 7.1: TI-530 PLC internal devices and instructions 73

Table 7.2: TI-530 PLC functions 74

Table 7.3: Memory addresses used as function operands 79

Table 8.1: Task codes for TI-530 PLC 81

Table 8.2: Memory areas and their corresponding task codes 85

LIST OF FIGURES

Figure 2.1: An electromechanical relay 7

Figure 3.1: Expert systems based control software generator for the PLC 20

Figure 3.2: General framework 22

Figure 3.3: General breakdown of a process 25

Figure 3.4: Basic form of functional code 31

Figure 3.5: Algorithm of the User interface module 40

Figure 3.6: General algorithm used to convert functional code to task code 42

Figure 4.1: Layout of drilling machine 51

Figure 4.2: Ladder Logic Diagram for example system specification . . . 58

1 INTRODUCTION

In a manufacturing facility, transfer lines, process control, robotics, material handling and flexible manufacturing systems all require some form of control to ensure their successful operation. Programmable Logic Controllers (PLCs) are used extensively today for manufacturing discrete system control. The PLC is programmed using Relay Ladder Logic diagrams. Writing and debugging programs for the PLC is a time-consuming and error-prone process and it accounts for a major component of the cost of implementing automated manufacturing systems. Such software is also incomprehensible and hard to maintain. To resolve such difficulties and to provide a high-level system of flexibility and maintainability, a knowledge-based control software generator may be used. Such a software automatically generates an executable computer code from a description of the control variables and the processes.

The purpose of this research is to develop a general framework of an expert system based automatic PLC control software generator that will generate the control task codes and the wiring instruction of a given PLC based on the user defined control function. The system will include:

1. User interface module

2. Functional code database

3. Wiring instruction database

4. Error checking module

5. Task code module

6. Task code database

The user interface module provides a friendly environment in which user may define the process to be controlled by giving the control variables, variable states, and control logic. This description is stored in a LISP-like syntax in functional code database. A wiring instruction is also generated to help the electrician identify the control system wiring. This is stored in the wiring instruction database. The user interface module is general in the sense that the functional code generated can be used with any PLC. The functional code is analyzed by the error checking module to see if programming rules of a particular PLC have been followed. If the functional code is error free it is retrieved by the task code module of the same PLC and converted into the task code which is stored in a task code file. The file can be downloaded into the PLC through serial communication. The task code module is a bidirectional module, which means that it may convert the functional code into task code and vice versa. The PLC control software generator is a generic system. With different error checking and task code modules, the system generates control software for any PLC.

In this research, a generic user interface module was developed that generates a functional code that is usable by error checking module and task code module for any PLC. The error checking and task code modules being system specific have been developed for the TI-530 PLC in the current research.

The rest of the thesis will be organized in four chapters, where the need for automatic control software generation, the use of expert systems in this area, and the automatic software generators developed so far are discussed in Chapter Two, the general framework of the control software generator and the implementation of the generator in this research are discussed in Chapter Three, and the results and conclusions of this research are discussed in Chapters Four and Five, respectively.

2 REVIEW OF RELEVANT LITERATURE

2.1 Historical Background

Control is the most important element in any automated production system. Transfer lines, numerical control, industrial robots, material handling and flexible manufacturing systems all require some form of control to ensure their successful operation [12].

The first form of industrial process control was manual regulation of production operations. The operator had absolute control. He had to use his own senses to determine how well the process was doing. The coming of measuring instruments eased the task of the operator, but the operator was still required to make necessary changes to the input variables to keep the operation running smoothly. The advent of the analog computer made this task even easier. Now the adjustments could be made automatically with very little human intervention for minor changes in the input variables. However, each "output variable" had to be monitored and changes were to be made in the "input variable" to maintain the output at the desired level. As industrial processes got more complex, more efficient means of carrying out the control actions were needed. The digital computer replaced the conventional analog control devices. The digital-computer regulated the process on a time-shared, sampled-data basis rather than by many individual analog elements,

each working in a continuous dedicated fashion. The digital computer offered not only an opportunity for greater efficiency in doing the same job than analog control, it also opened up the possibility for increased flexibility in the type of control action, as well as the option to reprogram the control action [12]. Many of the concepts and strategies used in conventional analog control are used today in computer control, especially in direct digital control.

Control can be divided into four categories:

1. Conventional linear feedback control (analog control)
2. Optimal control
3. Sequence control
4. Computer process control

Conventional linear feedback control is based on mathematical models. Linear differential equations are considered to be the building blocks of linear control theory.

Optimal control involves the use of a computer to calculate the optimum operating conditions at which to run the process [13].

Sequence control is concerned with coordinating the timing and sequencing of activities that take place in an automated production system. Sequence control consists of logic control and sequencing. Both types involve binary values, i.e., 0 or 1, on or off, high voltage or low voltage and so on. These control systems operate by turning on and off switches, motors, valves and other devices in response to operating conditions and as a function of time [12]. A

logic control system, also referred to as a combinational system, is a switching system in which decisions are made and actions are taken in response to events that occur in the production system. In other words, the outputs of the controller are determined by the inputs from the production system. A **sequencing system** uses internal timing devices to determine when the outputs to the production system should be changed. Thus sequential control provides output signals that are a function of instantaneous input variables as well as a function of time.

Computer process control uses stored program digital computer to control an industrial process. Computer programming for process monitoring and control is different from programming for data processing and scientific/engineering calculations in the sense that the computer must be capable of responding to:

Timer-initiated events - events triggered by clock time

Process-initiated interrupts - incoming signals from a process

Computer commands to process - the computer system must have the capability to direct the various process hardware devices that regulate the process in the desired manner

System and program-initiated events - events triggered by other computers in the network or by peripheral devices such as a card reader or printer

Operator-initiated events - events triggered by input from operation personnel [12]

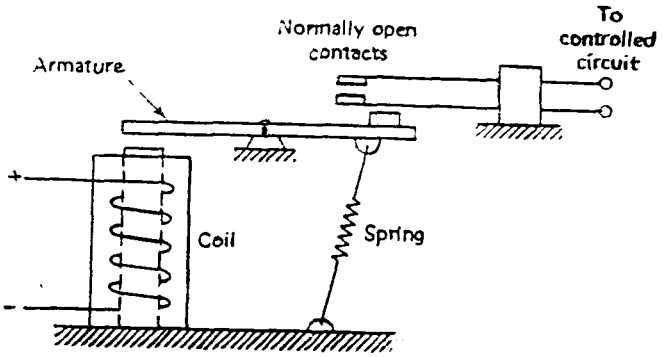


Figure 2.1: An electromechanical relay

Earlier, the operation of production equipment was controlled by means of conventional relay control systems. Relay control systems consist of electromechanical relays which are magnetic switches that can be actuated indirectly by other switches. The operation of an electromechanical relay relies on the use of electromechanical force generated by a coil magnet. When the coil is energized, an armature is drawn towards the coil. The armature is attached to a lever arm as shown in the Figure 2.1. The lever arm pushes against one of the contacts causing it to close against the opposite contact. Owing to the springiness of the contacts, they open when the coil is deenergized [13]. As can be seen relay panels were bulky, difficult to modify and time consuming (and hence expensive) to wire and document. Also, communication between a relay-control system and one of the newly emergent minicomputers for either data acquisition or supervisory control was impossible [32].

The year 1969 saw the birth of the Programmable Logic Controller (PLC). It was introduced in an effort to overcome the shortcomings of the conventional hard-wired relay control systems. Today, PLCs are used extensively for sequential control in transfer lines, robotics, process control and many other automated systems. It

is defined by the National Electrical Manufacturers Association (NEMA) as:

> A digitally operating electronic apparatus which uses a programmable memory for the internal storage of instructions for implementing specific functions such as logic, sequencing, timing, counting, and arithmetic to control, through digital or analog input/output modules, various types of machines or processes [12].

In essence, the programmable logic controller consists of computer hardware which is programmed to simulate the operation of the individual logic and sequence elements that might be contained in a bank of relays, timers, counters and other hard-wired components [12]. It is a sequential logic device which generates output signals based on the logic operations performed on the input signals. The program for the PLC consists of a sequence of instructions that determine the inputs, outputs and the logical operations.

Initially the PLC was successful due to its similarity to relay control systems. Even the programming methods used for PLCs - ladder logic diagrams, were very similar to circuit diagrams that the shop personnel were already familiar with. So the shop personnel did not have to learn an entirely new programming language in order to use the PLC. Programmable logic controllers have several advantages over the conventional relay controls. These are:

1. Programming the PLC is often easier than wiring the relay control panel.
2. The PLC can be reprogrammed. Conventional relay controls must be rewired and are often scrapped instead.
3. The PLCs take less space than relay control panels.

4. Maintenance of the PLC is easier, and reliability greater.

5. The PLC can be connected to plant computer systems more easily than can relays [12].

The PLC has six basic components. They are:

- Input module
- Output module
- Processor
- Memory
- Power supply
- Programming device

The input and output modules are connections to the industrial process to be controlled. The inputs consist of signals from limit switches, pushbuttons, sensors and other on/off devices. The outputs from the controller are on/off signals to operate motors, valves and other devices required to actuate the process.

The processor is the central processing unit (CPU) that executes various logic and sequencing functions. Tied to the CPU is the PLC memory which contains the program logic, sequencing and other input/output operations. A power supply provides DC power to operate the processor and I/O devices. The PLC is programmed by means of a programming device [12].

Basic control functions performed by the PLC are:

Control relay functions - an output signal from one or more inputs is generated according to a particular logic rule.

Timing functions - an output signal is generated a certain length of time after an input signal is received or an output signal is maintained for a certain length of time.

Counting functions - the number of input contact closures are added up and an output signal is produced when the sum reaches a certain count.

Arithmetic functions - addition, subtraction and comparison.

By definition of computer process control programmable logic controllers can also be used for computer process control because the processor of the PLC is a stored program digital computer. The capabilities of programmable controllers have evolved so that they can perform much of the data-processing and other functions that were previously reserved for computer control.

2.2 Expert Systems in Industrial Control

Knowledge-based expert systems are the latest addition to the rapidly developing field of software automation [16]. The main reason behind this is that expert systems contain the knowledge of a human expert. They are suitable for an industrial environment where system's configuration frequently changes according to the model changes of products or enhancements in the system [30] [31]. Knowledge-based programmable controllers have two distinct advantages over the conventional programmable controllers. They are more flexible and versatile - flexible because unlike the conventional controllers, the knowledge-based controllers are programmed

by telling the controller how the plant works and specifying the process requirements and versatile because the system has the knowledge of the way in which the plant works and the process requirements rather than just a program [27].

In highly automated factories of today control is very complex and involves several steps:

- planning
- scheduling
- real time scheduling and monitoring
- coordination
- control

Expert Systems are already being used successfully in the first three areas. Work is now being done towards their use in coordination and control [2].

Tashiro, Komoda, Tsushima and Matsumoto developed a Rule-Based Control Software. This software was developed for constraint combinational control of discrete event systems, especially factory automation systems. Here control logics are embedded in software as data and control actions are automatically inferred from the system situations by using production system methodology. Signals from/to sensors/actuators are input/output through a bit-table. Process interface definition defines correspondence between each bit in the bit-table and a condition or a conclusion described by rules. If-then rules and process interface definition data are input/updated through a console CRT using a rule editor. A rule-structure checker and a rule simulator are used to check rule correctness [30] [31].

This software approach involves direct control of the manufacturing process. In a lot of cases, this would mean replacement of the previous methods used for process control. For instance, to use the Rule-Based Control Software in a factory that is currently using a PLC for process control would mean removing the PLC and bringing in a new computer system that would use expert systems to control the process. This may not be very economical.

2.3 Automatic Code Generators

The process of control software developed can be simplified if such code can be generated automatically. Constructing control software by conventional methods such as the use of general purpose programming languages is an expensive process. Such software is hard to understand and maintain and requires extensive time-consuming debugging. Automatic generation of a control program would require that the control logic be specified and an executable control program is generated from the control logic specification. Transformation of a specification into an implementation eliminates the highly error-prone and programmer dependent task of generating executable code [24]. Other functions may also be added into the automatic code generator package, such as modules to verify, analyze and simplify the control logic specification.

One of the earliest attempts at automatic control software generation was made by Aldana, Peire, Penalver, and Uceda. They developed a method for computer aided generation of microprocessor software for controlling static power converters. Given the system specification, the elementary states that define all actions to be done in every moment and the necessary control matrices are generated [1].

Hanselmann and Schwarte developed a system for automatic production of efficient code for a multivariable controller to be run on fast target processors such as digital signal processors (DSP) and 16 bit microcontrollers and microprocessors. The code is efficient in terms of runtime memory usage and programmer productivity. Given a system specification, which is hardware independent, and a description of the target hardware, an assembly language source code is generated that is necessary to control the described system [14].

Automatic code generators have inspired the work done on electronic engine control strategy at the Ford Motor Company. Srodawa, Gach, and Glicker developed a code generator for the Ford/Intel 8061 microprocessor. The description of the control algorithm is coded in a high level application language called Ford Automotive Control Terminology (FACT). The complete FACT specification is converted to DEC-VAX machine code [28].

Code generators have also become available in the field of digital filtering. A system written in MACSYMA generates FORTRAN code that is capable of solving stochastic control and nonlinear filtering problems in symbolic form. The system uses a list of keywords and pieces of data that describe the problem. This list is analyzed and a new list is built that consists of instruction in an intermediate language called Macrofort for macro-FORTRAN. Finally the list is translated in a FORTRAN 77 program [20].

Another code generator was developed at Carnegie Mellon University for automatic programming of controllers for discrete manufacturing processes. The control logic is specified in terms of discrete states of physical states required for the operation. This specification is given in terms of a rule-based specification. The computer

integrates the control logic for the whole system, analyzes the control specification for completeness and logical consistency, and then generates a C source code for on-line control computers. The functional description of the manufacturing process and the process control logic is entered into a relational database at the work station. A petri net model of the system is constructed automatically from the given high level specification. This petri net model is analyzed for completeness and logical consistency. Finally, it is this petri net model that is translated into an executable control code. This code is then ported from the work station to the computer that controls the process [24].

Expert Systems can be used in the development of automatic code generators. A code generator similar to the one developed by Krogh, Wilson and Pathak was developed for stochastic control and filtering problem [10]. An expert system was used for automatic FORTRAN code generation. The system is written in MACSYMA and Lisp. It has four major components:

1. A modular system of programs written is MACSYMA which solve stochastic and nonlinear filtering problems in symbolic form.

2. A natural language interface.

3. A "theorem proving module" using PROLOG capable of checking the well-posedness of linear and nonlinear partial differential equations specified in symbolic form

4. A module for generating FORTRAN code from the symbolic manipulation module of the system

The function of the system is to accept input from the user in natural language with model equations expressed in symbolic form, to automatically select a solution technique for the control or filtering problem, to reduce the model equation by symbolic manipulations to a form appropriate for the technique, checking the well-posedness of the model along the way; and to automatically generate a numerical language code realizing the solution algorithm.

Different methods have been used to specify the control logic of the system to be controlled among which a high level specification language and petri nets are the ones most widely used. Petri nets are a well known powerful and rigorous tool to describe distributed systems, featuring concurrent processes and synchronization problems. The Petri net approach for detection and handling purposes and the expert systems approach for diagnosis and decision are strongly similar and can be utilized together [25] [2]. An expert system can be used to model the individual processes or events and petri nets can be used to model the relationships between the processes [35].

Gentina and Corbeel used Structured Adaptive Colored Petri nets (S.A.C.P.N) and artificial intelligence techniques to automate the control design of flexible manufacturing systems [8]. This is done in four steps:

1. Specifications of the requirements and objectives of productions in terms of rules, facts and procedures are given according to an expert system modeling.

2. From the system specification the first global colored petri net is deduced by inference.

3. Different solutions are proposed. For each solution proposed, the basic global

scheduling of the process in terms of colored interpreted petri nets and the strategies of control from interpreted meta-rules are obtained.

4. From this hybrid model composed of both strategy meta-rules and colored petri nets a more complete model is deduced which gives both a structured decomposition of the control process and a precise definition of links between first the elementary tasks at the level of an S.A.C.P.N. and of the links between S.A.C.P.N. model and the strategy selected at the hierarchical level. This model gives a possible procedural implementation of the process control on a set of computers.

An intermediate control language called Functional Control Language (FCL) was developed for improving the software portability of PLCs [29]. FCL is and intermediate language situated between PLC programming languages and PLC machine code. It has a Lisp like structure and can be converted from and into each PLC programming language such as Ladder Logic Diagram, Functional Block Diagram or Sequential Functional Chart. A system called FAISES (Fuji AI-based Software Engineering System) was developed to:

1. convert PLC language to FCL and vice versa.

2. generate machine code and convert this machine code back to FCL, if possible, depending on the target machine language.

3. accumulate and select software libraries.

4. generate various kinds of documents such as program diagram, program configuration diagram, I/O allocation list, memory map, cross-references list, etc.

This FCL may be used as a high level specification language to specify the control logic of the system directly instead of translating it from the PLC programming language. The user defines the system to be controlled in terms of the control variables, variable states, and control logic. This specification is stored in FCL. Next, this FCL is translated to machine code.

Except for the automatic code generator developed at the Ford Motor Company, all the other code generators described generate control code in a high level language such as FORTRAN, C, assembly language, etc. An attempt will be made in this research to develop an expert system based code generator that takes as input a description of the process to be controlled and generates the PLC task code directly. The task code is the machine code representation of the ladder logic diagrams. Thus, the interface needed for translating the high level language control code to the machine code acceptable by the process controller is eliminated. Along the way wiring instruction is also generated to help the electrician identify the control system wiring. The control logic description is stored in an intermediate functional code similar to the one described above [29]. It is this intermediate functional code that is translated to the task code for the TI-530 PLC. Given a task code, this task code module can also translate the task code back to the functional code representation of the control logic. Thus the task code module is bidirectional. Use of the functional code improves the software portability of the PLCs. This task code can easily be enhanced to translate the intermediate functional code to the task code for other PLCs as well.

3 METHODOLOGY

A control software generator was developed in this research. The process of control software generation involves three main steps.

1. Description of system to be controlled
2. Analysis of the system specification for syntax errors
3. Conversion of system specification to task codes

The user develops a description of the system to be controlled. The user provided control system description is converted to a formal specification. This formal specification, before being converted to the machine code representation, i.e., task codes, is analyzed for syntax errors. This chapter discusses the general structure and the implementation of the automatic control software generator in detail.

3.1 General Structure of the Software Generator

3.1.1 Introduction

The different steps of development of control system specification, syntax error checking, and translation of control system specification to the task code are done in sequence. These steps of control software generation are performed using a modular

approach with each module performing a specific task. The software generator consists of a total of six modules:

1. User interface module
2. Functional code database
3. Wiring instruction database
4. Error checking module
5. Task code module
6. Task code database

A pictorial representation of this framework is shown in Figure 3.1.

The user interface module, as the name suggests acts as an interface between the user and the task code module. It simply translates the user given control system description into an intermediate functional code specification. This functional code is a more formal description of the same system. After the functional code is generated, the user interface module also creates a wiring instruction. This wiring instruction provides an aid to the electrician by showing the connections between:

- various contacts in the input module of the PLC and the input devices
- various coils in the output module of the PLC and the output devices to be controlled

The functional code and the wiring instruction are stored in their respective databases. The user interface module is generic, in the sense that it can take any system

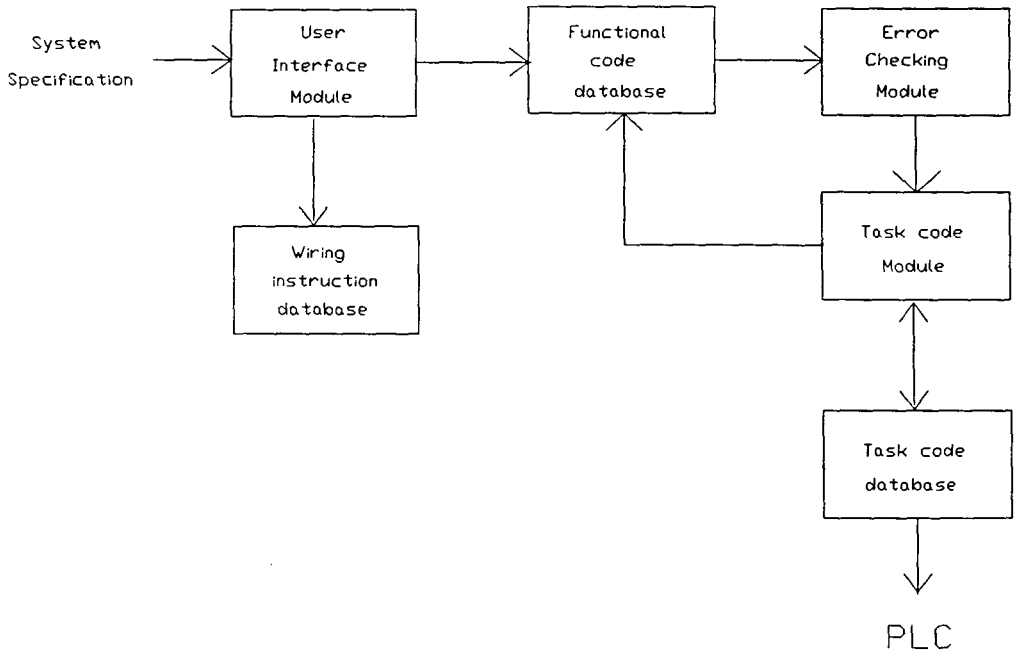


Figure 3.1: Expert systems based control software generator for the PLC

specification or process description and translate it into a general functional code description.

The functional code is retrieved by the error checking module. The error checking module is system specific, which means that one error checking module is needed for each different type of PLC since PLCs vary in their programming design. The error checking module simply performs syntax checks. If the system specification is error free it can be input into the corresponding task code module. The task code module, which is also system specific, translates the given functional code to the task codes for that particular PLC. The task code module is bidirectional, which means that not only can it translate the functional code into the corresponding task codes, but it can also translate the given task codes into the corresponding functional code specification. The advantage of it being bidirectional is that if the user modifies the task codes, then the user would not have to redefine the control system description to get the functional code. The functional code could easily be obtained from the task codes directly. These task codes can be downloaded to a PLC for execution through serial communication. If the PLC programming device is connected to the PLC then the ladder logic diagram corresponding to the task codes can also be checked.

Having an intermediate functional code improves the portability of the PLC software. The PLC programming languages depend greatly on the hardware of each PLC, but the functional code that has been developed in this research is machine independent. Thus the functional code can be used with any PLC and one does not have to reprogram the whole control system in order to use a different PLC. A more general framework showing this portability is shown in Figure 3.2.

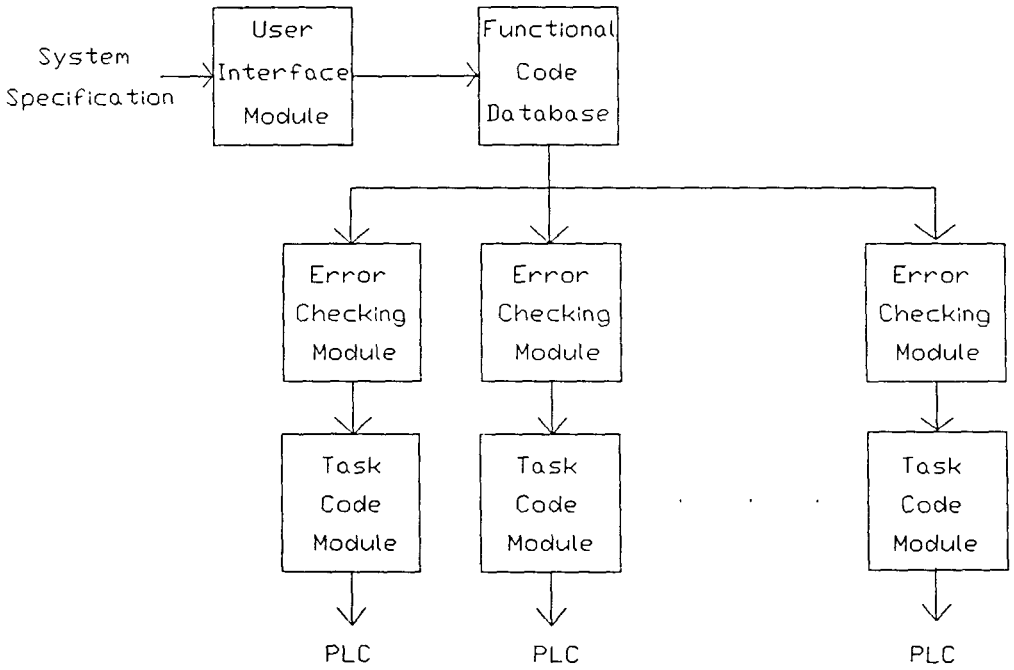


Figure 3.2: General framework

The general framework in Figure 3.2 shows one generalized user interface module and one functional code database which are hardware independent and one error checking module and task code module each for each different type of PLC. Thus to build a complete system several error checking and task code modules are needed. Once a general functional code is generated it can be used with any desired PLC, thus improving the portability of PLC software.

3.1.2 The user interface module

The user interface module is a generic module. It translates the user given description of a system or process to be controlled to an intermediate functional code specification.

In order to be able to define a process or system that one wishes to control, it would be necessary to understand what the process is, what it constitutes of, and how it is to be described. The following sections describe a process, its general breakdown structure, and the format to be followed when entering the process description.

3.1.2.1 Process description By dictionary definition a process is a systematic series of actions. Stamping of parts, putting parts in a bin, drilling holes in parts etc. are all different manufacturing processes. As shown in Figure 3.3 a process can be broken down into a several control steps, namely operations. Thus an operation is just one control step of the entire process. An operation can be further decomposed into a set of conditions and actions. This decomposition can be best illustrated through an example. Consider a process of filling a tank with wa-

ter. This process would require one switch to turn on the pump to fill the tank, one pushbutton switch to stop the pump in case of an emergency, and one limit switch to test if the tank is full or not. This process can be broken down into operations.

1. When the pump is turned on, the emergency switch is off and the tank is not full a solenoid is turned on to start filling the tank.

2. As long as the solenoid is on, the emergency switch is off and the tank is not full the water keeps flowing into the tank. When the tank is full the limit switch is turned on and the water will stop flowing.

The various on/off switches are the input devices and the on/off state of the switches constitute the conditions of the operations involved. The solenoid used to start and stop of the flow of water is the output device. The starting and stopping of the flow are thus the actions involved. When the specified conditions of an operation are true, the corresponding action is performed.

Conditions - The conditions of an operation consist of **physical conditions** and **functions**. Physical conditions are associated with the physical state of discrete (on/off, closed/open) input states such as limit switches, photodetectors, pushbutton switches, internal control relays, and circuit breakers. Table 3.1 shows the one/zero interpretation for different input and output devices.

Sometimes it may become necessary to perform a function in order to control an operation. For example, a process of filling a bin with say 20 parts requires some way in which parts can be counted. This would require a counting function. The following are the different types of functions that may be used in controlling a process:

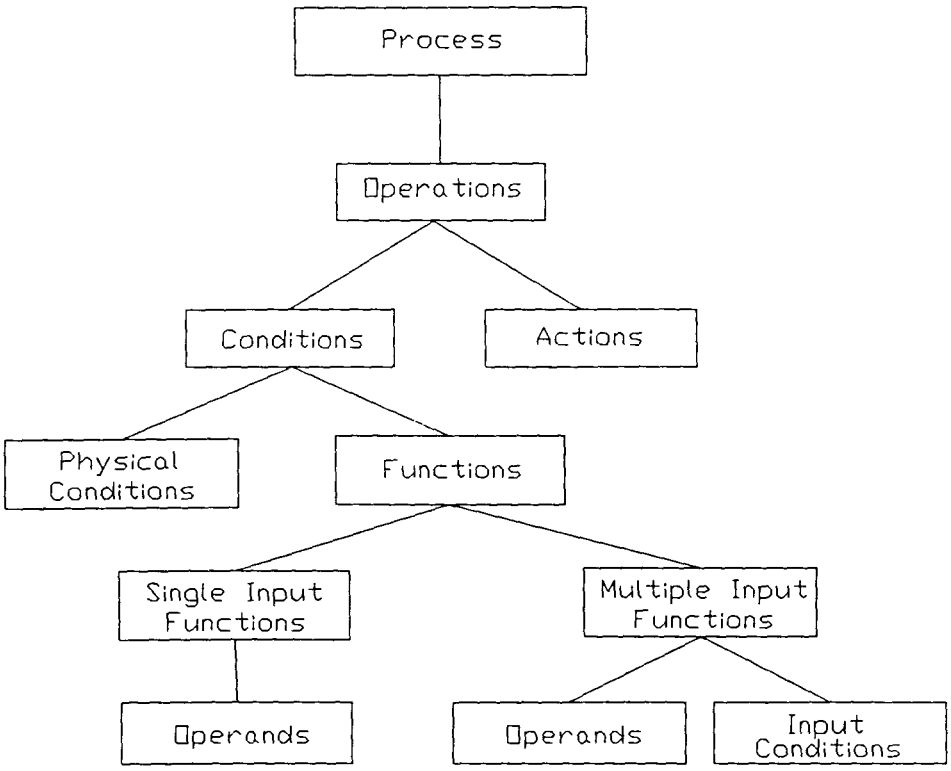


Figure 3.3: General breakdown of a process

Table 3.1: Binary input and output devices

Device	One/Zero Interpretation
Input	
Limit switch	Contact/no contact
Photodetector	Contact/no contact
Pushbutton switch	On/off
Control relay	Contact/no contact
Circuit breaker	Contact/no contact
Output	
Motor	On/off
Alarm buzzer	On/off
Control relay	Contact/no contact
Lights	On/off
Valves	Closed/open
Clutch	Engaged/not engaged
Solenoid	Energized/not energized

1. Timing functions - An example would be the generation of an output signal after a certain length of time has elapsed. Another example would be to maintain an output signal for a certain length of time.

2. Counting functions - The counter counts up the number of input contact closures and produces a programmed output when the sum reaches a certain count. The counter can then be reset.

3. Arithmetic functions - These includes capabilities of adding, subtracting, comparing, dividing, multiplying and finding the square root.

4. Bit functions - These functions involve clearing a particular bit, testing the status of a selected bit, setting a particular bit to 1, etc.

5. Word functions - These functions involve converting binary to decimal, converting decimal to binary, ANDing two words in memory, ORing two words in memory, rotating a word in memory, performing exclusive OR of two words in memory, etc.

6. Move functions - These functions involve loading a data constant to a memory location, moving a set of words starting at a particular memory location to another memory location, etc.

There are two types of functions - **single input functions** and **multiple input functions**.

- Single Input Functions require only one input condition. When the physical condition is true the function is performed. A function that is used to add numbers in two memory locations is an example of a single input function. When the given input condition for the function is true the addition is performed.

- Multiple Input Functions require one or two additional input conditions. These additional input conditions may be required to reset the function operands and/or enable the function. A counting function would be an example of a multiple input function because an additional condition would be needed to enable the counter. The counter counts the number of times the initial input condition is true. It counts until the sum reaches a certain preset count. When the additional condition is true the counter is enabled to start counting. When the enable input is turned off the counter is reset to zero.

Actions - The actions of an operation are the alternation of output device status or internal instructions. The output devices include devices such as motors, alarm buzzers, lights, valves, clutches, and solenoids and internal devices and the instructions are internal control relays, master control relays, unconditional end, conditional end, etc.

3.1.2.2 Format of the process description To describe the process to be controlled it must be broken down according to the structure described above. The system must be described in a fixed format and must be entered in an ASCII file.

The process is described in terms of operations using some keywords. A total of seven keywords are used in order to identify the different subdivisions of an operation. There are several attributes associated with each keyword. These attributes are specified in fields. Table 3.2 shows the keywords and their attributes. Table 3.3 shows the order in which the keywords are used to describe the operation.

Each operation is identified by an operation name and a unique operation number. The operation name is specified using the keyword 'Operation_name' followed by the name of the operation and the operation number is specified using the keyword 'Operation_number' followed by an operation number. The operation number must be unique in order to distinguish between different operations in a process. For example, an operation for filling a water tank may be named and numbered as follows:

Operation_name Fill_tank

Operation_number 1

The physical conditions and functions of an operation are specified using the keywords 'Physical_condition' and 'Function' respectively. The physical condition is specified using its keyword, the device name, and the value the device is to take 0 or 1. If the physical condition is a control relay a control relay number must also be given by the user. For instance, a physical condition consisting of 'Limit_switch_1' that must be closed and another physical condition consisting of 'Control_relay 10' that must be off are specified as follows:

```
Physical_condition    Limit_switch_1  1
Physical_condition    Control_relay   0   10
```

The functions involved in an operation are specified using the keyword followed by the name of the function and the operand list. In the case of the multiple input function the conditions used for resetting and/or enabling the function are specified using the keywords 'Function_condition_1' and 'Function_condition_2'. Functions may require both resetting and enabling conditions and hence a function may have upto two types of input conditions for resetting and enabling functions. To make the function logic concise a restriction has also been placed on the number of input conditions of each type. Any multiple input function may have upto two input conditions of each type. In most cases, it is sufficient to have upto two input conditions of a particular type and hence this restriction should not pose a problem. For example a counter function has one operand which is a preset count, say 20. The input condition required to enable the function is 'Enable/Reset_Limit_switch_2'. Each

time 'Increment_Limit_switch_1' is turned on the counter is incremented. When 'Enable/Reset_Limit_switch_2' is open the counter is reset to zero. This function is described as follows:

Physical_condition	Increment_Limit_switch_1	1
Function	CTR	20
Function_condition_1	Enable/Reset_Limit_switch_2	1

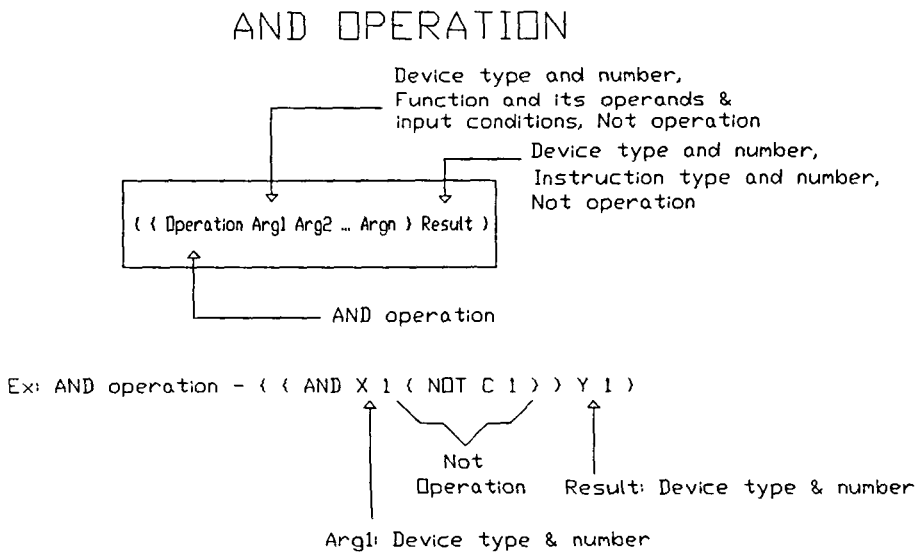
For more detail on the order of the function operands and the reset and enable conditions refer to Table 6.4 in Appendix A.

The actions of an operation are specified using the keyword 'Action' followed by the name of the output device and the value the device would take if the conditions for the operation are true. If an internal device is used a number is also included in the description. For example an action that turns a 'Control_relay 45' on and an action that turns a solenoid on are specified as follows:

Action	Control_relay	1	45
Action	Solenoid	1	

See Appendix A for a more detail listing of the rules that must be followed when describing a control process.

3.1.2.3 Description of the functional code The functional code follows a Lisp format. It uses a set of curly brackets to represent one unit, shown in Figure 3.4. Each operation, consisting of physical conditions, functions if any, and actions is enclosed in a set of curly brackets.



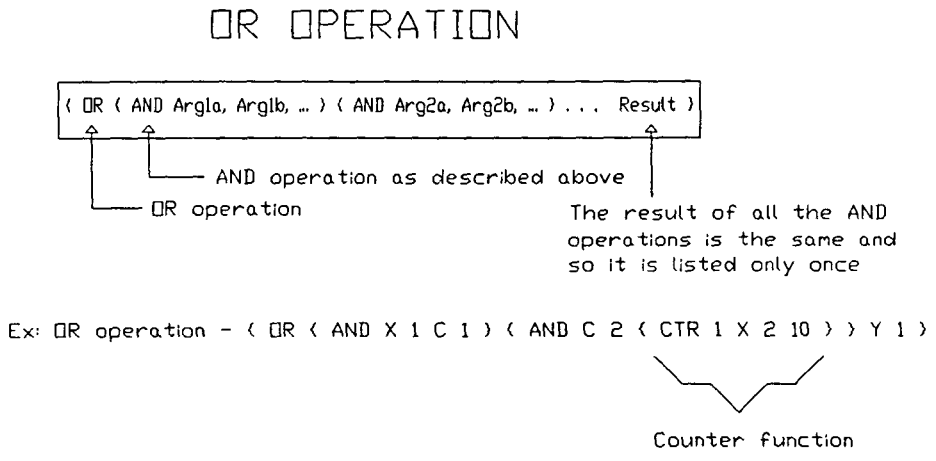


Figure 3.4: Basic form of functional code

The physical conditions, functions, internal devices and instructions, and actions are represented by a symbol and a number. There is a different symbol for the input devices, output devices, internal devices, internal instructions, and functions. See Table 3.4 for the symbols used. To distinguish between devices, internal instructions or functions of the same type a number is used. A device with a value of 1 is represented with a symbol for the device and a number, whereas a device with a value of 0 is represented with a reserved word 'NOT', a symbol, and a number in a set of curly brackets. For example an input device with a value of 1 is represented as X 1, and one with a value of 0 is represented as { NOT X 1 }. A function and its operands are included inside another set of curly brackets, thus treating it as a subunit.

All the physical conditions of an operation are ANDed together. Some of the operations may have the same action. All such actions are ORed. Thus the functional code mainly consists of AND and OR operations. Each AND operation is enclosed in a set of curly brackets and is thus treated as one unit.

Looking back at the water tank example, the first operation involves the switch to turn the pump on or off, the emergency switch, and the limit switch. These three devices are treated as input devices They are labeled as X 1, X 2, and X 3 respectively. The conditions of this operation are:

1. X 1 must be on (pump must be on)

2. X 2 must be off (emergency switch must be off)

3. X 3 must be open (tank is not full)

The action of this operation uses a solenoid. This solenoid is labeled Y 1. The

action involves turning on the solenoid. If the above mentioned conditions are true the solenoid represented by Y 1 is turned on.

The second operation has the solenoid (Y 1), the emergency switch (X 2), and the limit switch (X 3) as parts of the physical conditions. The conditions involved are:

1. Y 1 must be on (solenoid must be on)
2. X 2 must be off (emergency switch must be off)
3. X 3 must be open (tank is not full)

As long as these conditions are true the solenoid Y 1 is to stay on, i.e, the water is to continue flowing. Both the above mentioned have the same action of keeping the solenoid Y 1 on. Hence these two operations may be ORed. Thus the functional code for this system is as follows:

{ OR { AND X 1 { NOT X 2 } { NOT X 3 } }
{ AND Y 1 { NOT X 2 } { NOT X 3 } } Y 1 }

3.1.3 The wiring instruction

The user interface module generates a wiring instruction after the functional code is generated. The wiring instruction acts as a link between the functional code and the control system description in the sense that it shows what the symbols of the input and output devices represent. It simply lists the symbol and the number used in the functional code and the corresponding device that it represents. For instance, in the water tank example shown above the wiring instruction would be:

X 1 Pump_On/Off switch

X 2 Emergency_switch

X 3 Water_Full_Level_Limit_switch

Y 1 Fill_tank_solenoid

The wiring instruction is stored in the wiring instruction database.

3.1.4 The error checking module

PLCs vary in their programming styles. Each different type of PLC has its own programming rules. In order to be able to use the functional code with a particular PLC it must be made sure that it does not violate any of the programming rules of that particular PLC. Hence before the functional code can be translated into the task code some error checking must be done. For instance, some PLCs may have a restriction on the number of physical conditions or the number of actions an operation may have. As it can be inferred from the above description, the error checking module is system specific. Hence one error checking module would be needed for each different type of PLC.

The error checking is mainly syntax checking. Error checking must be done to see if programming rules are followed, if the functions used are available on the PLC, if functions have the required operands in the required sequence, etc. The errors found may be written to a file and this file may be used to correct the indicated errors.

3.1.5 The task code module

The task code module translates the functional code generated by the user interface module to the task codes of a particular PLC. Task codes are the executable machine code representation of the control logic specified by the user. The task codes may be ASCII code or hexadecimal or any other type that is machine readable. The task code module is also capable of converting the task codes back to the functional code. Thus if modifications are made to the control logic at the task code level, the corresponding functional code can be obtained directly from the task codes rather than having the user redefine the system specification and generating the functional code from the user interface module.

Each different PLC has a different machine code representation for the control logic. Hence the task code module must generate a task code representation that is acceptable by a particular PLC. In other words, the task code module is system specific. One task code module is needed for each different type of PLC.

3.2 Implementation

The automatic control code generator has been developed on the VAX/VMS operating system. The user interface, the error checking module, and the task code module have all been written in OPS5.

3.2.1 The user interface module

The user interface module takes as input a control logic specification that is given by the user. The system specification is to be entered following the format

that has already been described in the previous section.

Each operation is read in sequentially. As each physical condition is read in, it is determined if it includes an input device, a control relay, or an output device. Depending on the type of physical condition a symbol and a number is assigned to it (see Table 3.4 for the different types devices used). In the case of input and output devices a sequential number is assigned to them, whereas in the case of control relays the number given by the user is assigned.

If the device (other than control relay) is read in as part of a physical condition, it is first checked to see if this device has been used before in a previous operation. If not, it will be treated as an input device and is assigned the next number in sequence. If this device is read in the future it will always be treated as an input device and will always be assigned the same number. For instance, if 'limit_switch_3' is read in for the first time as part of a physical condition and there already exists another input device then 'limit_switch_3' is assigned the symbol X and number 2 (since it is the second input device in sequence), and in the future use of 'limit_switch_3' it is always referred to as X 2. If the device read in has been used before then it is used in the same context as before with the same number. A device read in as part of an action is treated in the same way. However, it does not make sense to have an input device (type X) as a part of an action. Hence care must be taken in the sequence in which the operations are entered. In general the sequence of operations is not important, except in one case. If a device must be treated as an output device, then the operation that lists it as an output device, i.e., as a part of an action, must be entered before the operation in which the output device is a part of a physical condition.

Some of the common functions that PLCs are capable of handling are shown in Appendix A. The function names and operand sequence are fixed. Along with a name and operand list multiple input functions have input conditions associated with them. The input conditions are similar to physical conditions except for the fact that they have a special purpose, i.e., to reset or enable a function. So the input conditions are assigned a symbol and number in the same fashion as the physical conditions. If a multiple input function has two of any one type of input condition, the situation is handled as follows:

1. Suppose a multiple input function has two input conditions under which it is reset, say two different limit switches must be closed. As a first step a new operation is created that uses these two limit switches as input devices. A control relay is used as an output device for this operation. It is turned on whenever the physical conditions corresponding to these two limit switches are true.

2. This control relay is then used as the input condition to the function.

Thus a new operation is added to the process, but this operation is internal to the PLC. The input conditions are then finally included in the operand list of the function. For example: A counter function requires one input condition to enable the counter. Suppose two limit switches, 'Enable/Reset_Limit_switch_1' and 'Enable/Reset_Limit_switch_2', must be closed to enable the counter. These two switches must be open to reset the counter to zero. Each time 'Control_relay 5' is turned on the counter increments by one. The description for this operation would be:

.

.

Physical_condition	Control_relay	1 5
Function	CTR	15
Function_condition_1	Enable/Reset_Limit_switch_1	1
Function_condition_2	Enable/Reset_Limit_switch_2	1
Action	Conveyor_On/Off_Solenoid	1

Since the operation has two input conditions of the same type, a new operation is created that uses the two limit switches as physical conditions and if the two limit switches are closed a randomly chosen control relay is turned on. Thus the control relay is part of the action of this new operation. The functional code for these two operations would be as follows:

{ { AND X 1 X 2 } C 43 }

{ { AND C 5 { CTR C 43 15 } } Y 1 }

The wiring instruction is generated after the functional code has been generated. The wiring instruction simply shows the relationship between the input devices used by the user and the symbols used in the functional code. Thus the wiring instruction would help the electrician in making connections between the input (output) module of the PLC and the input (output) devices used. For the above counter example the wiring instruction would be:

X 1 Enable/Reset_Limit_switch_1

X 2 Enable/Reset_Limit_switch_2

Y 1 Conveyor_On/Off_Solenoid

The algorithm used is this module is shown in Figure 3.5.

In this research the functional code generated was used with the TI-530 PLC. The user manual to be used for TI-530 PLC is given in Appendix B. It lists the functions available on this PLC along with the types of addresses and their ranges that may be used as function operands.

3.2.2 The error checking module

The Error checking module performs syntax error checks on the functional code that is to be used on the TI-530 PLC. The syntax rules for TI-530 PLC are listed in Appendix B. These mainly include checking the number of physical conditions, actions, single input functions, and multiple input functions in an operation and the sequence in which these may be entered. Detailed error checking is done on functions to make sure that the operands are in correct sequence and the correct memory addresses are used. All the errors found are written to a file. This file may be accessed to see what the errors are. A summary of the errors is output to the screen to give the user the information regarding the number of errors found. If any errors are found, they must be corrected before the functional code can be used on the task code module. If a task code with errors is entered into the PLC it may result in a fatal error. The PLC will not function but there will be no way of determining what the error is once the task code has been downloaded into the PLC.

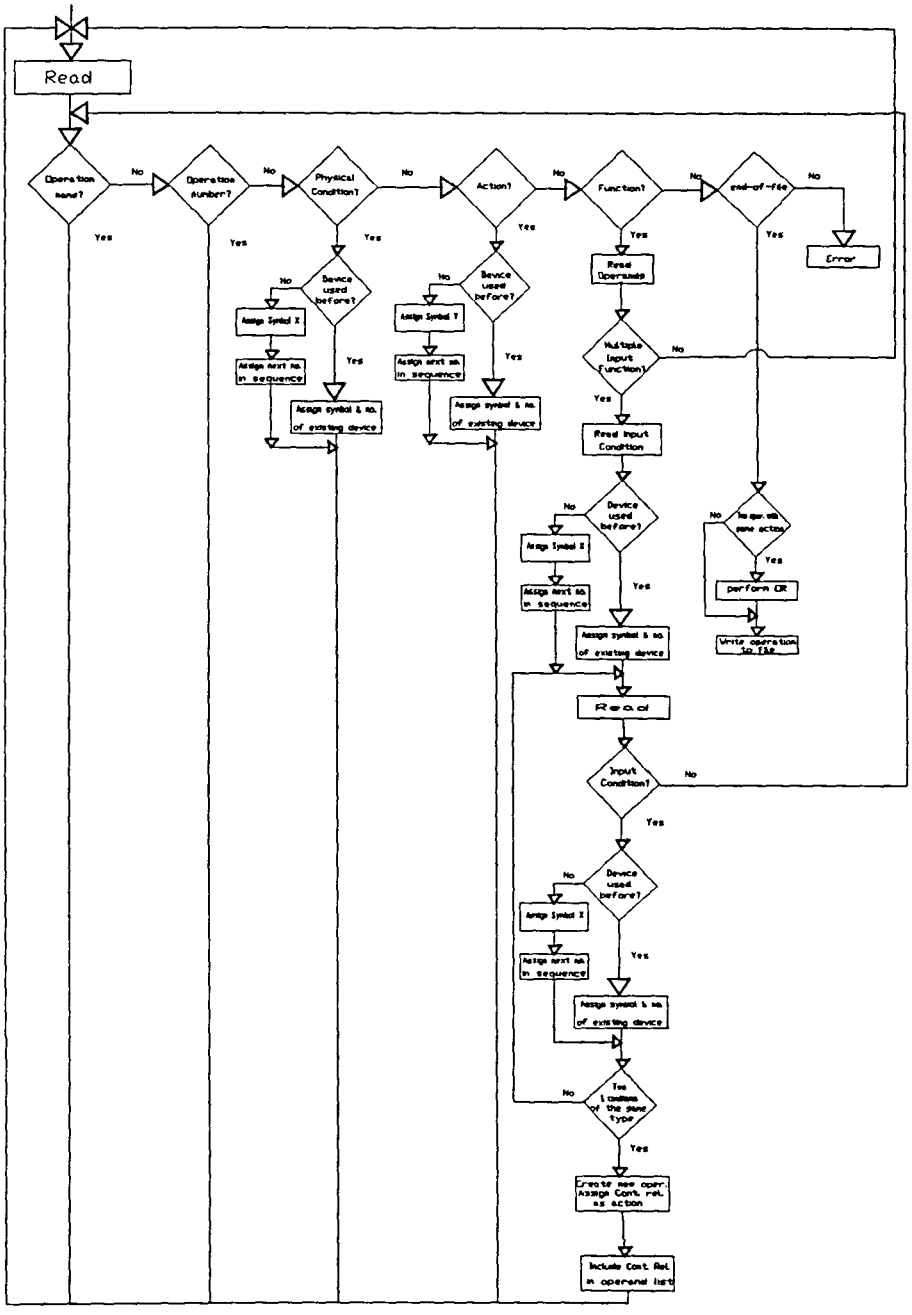


Figure 3.5: Algorithm of the User interface module

3.2.3 The task code module

The task code module has been developed for the TI-530 PLC. It converts the given functional code to the task codes and vice versa. It provides the user with two options:

1. to convert the functional code to the corresponding task codes, and
2. to convert the task codes to the functional code specification.

3.2.3.1 Conversion of functional code to task code The functional code is read in sequentially, one word at a time. As each word of the functional code is read in it must be determined if it is an input device, an output device, an internal device, an internal instruction, a function, or a memory address location. There is a different task code for each of these different devices, instructions, and functions. The task codes for the physical conditions and the input conditions of multiple input functions are the same. The general algorithm used to perform these checks is shown in Figure 3.6.

The task codes of all devices, instructions, and functions are listed in the order in which they are listed in the functional code except in the case of multiple input functions. In the functional code the order is as follows:

function name

input condition(s)

operands

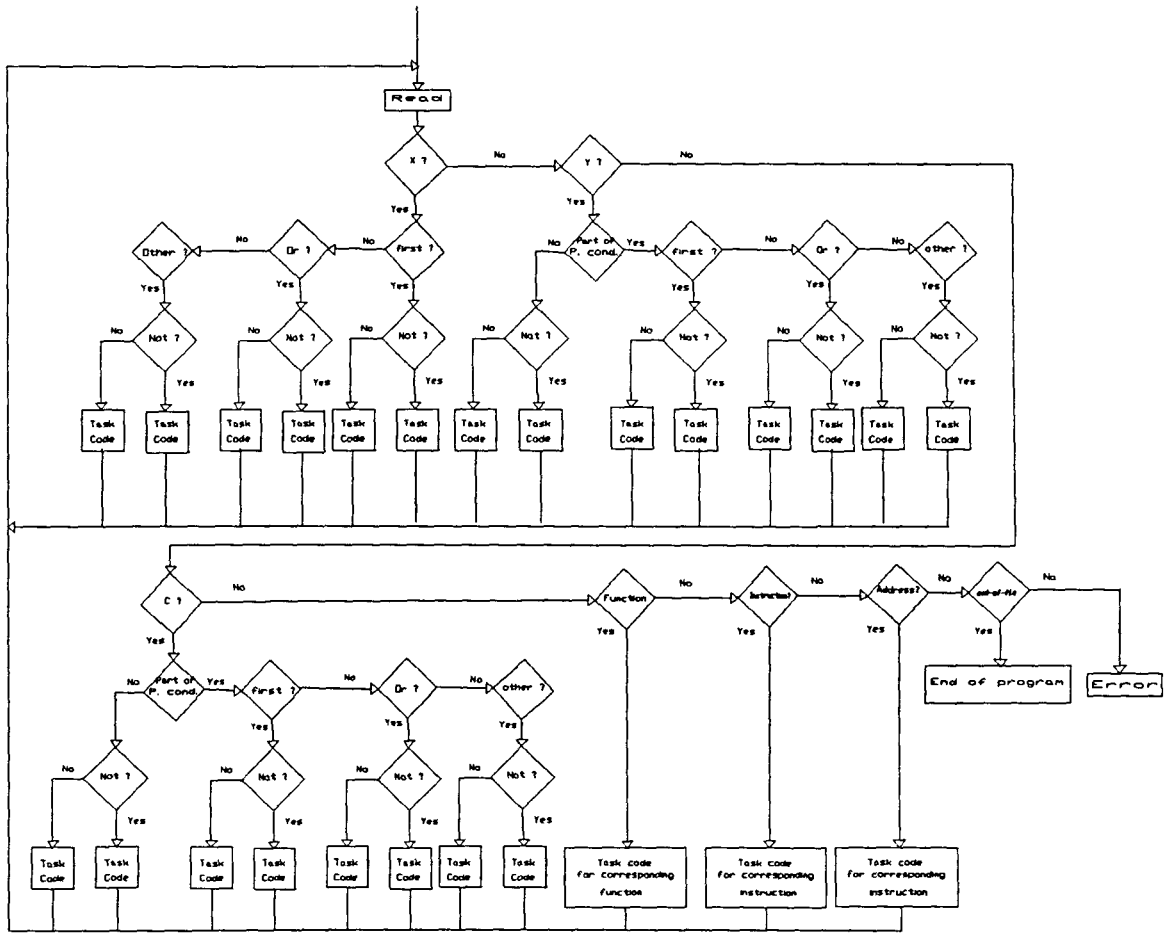


Figure 3.6: General algorithm used to convert functional code to task code

On the other hand the task codes for the function and its operands are listed in the following order:

input conditions

function name

operands

For example, the multiple input function, counter (CTR) has one input condition and one operand (see Appendix A). The functional code for this function would be:

.....{ CTR 1 X 2 25 }.....

where X 2 represents an input device which must have a value of one in order to enable the counter and 25 is the preset counter value that the counter counts up to. The task codes would be:

.....8502 A801 0019.....

where the task code 8502 represents the input device X 2, the task code A801 represents the counter numbered 1, and the task code 0019 is the hexadecimal representation of the preset value 25. As can be seen the position of the function name and the input condition is interchanged.

The task codes for the TI-530 PLC are listed in Table 8.1 in Appendix C. The task codes are four digit hexadecimal numbers. In the case of devices, functions and instructions, the two most significant digits simply represent the type of device,

function, or instruction and the two least significant digits indicate the device, function, or instruction number in hexadecimal. In the case of memory location addresses the most significant digit represents the starting memory location and the last three least significant digits indicate the memory location offset. For example the task code for an output device numbered 14, i.e., Y 14 used as part of an action is 990D where 99 is the numeric symbol for an output device used as an action in an operation. The task code for memory address V500 is 01F3 where the most significant digit zero indicates the starting address 0000 and 1F3 is 500 in hexadecimal and it is the offset from the starting location.

3.2.3.2 Conversion of task code to functional code The conversion of the task codes to the functional code follows the same algorithm as for conversion of functional code to task code. Each task code read is analyzed to see what it represents and it is then converted to the corresponding device, instruction, function, or memory address. The algorithm is straightforward except in the case of multiple input functions. Since the task codes for the input conditions are listed before the task code for the function and since the task codes for the input conditions and physical conditions are the same, there is no way to know beforehand that the task code read in is that of an input condition or a physical condition. The only way that it can be determined that an input condition was read in is when on further reading a multiple input function is encountered. The task code is initially read in as a physical condition. On reading the task code of a multiple input function one has to back up and retreat the previously read one or two physical conditions (depending on the type of function) as input conditions and copy them into the

function operand list. For example when translating the above task codes of the counter function, as 8502 is read it is translated into an input device which is a part of the physical condition. When the next task code is read and it is determined that it represents a multiple input function the input device read in prior to the function is now retranslated into an input condition of the counter function.

3.2.4 Some general comments on the user interface module and task code module designs

Expert systems are not well suited to perform problems that involve a complex numerical calculations and string manipulations. Hence, to do such tasks in this research external routines have been written. A total of six external routines have been used which are described as follows:

1. Rand - a random number generator

2. Convert - a string manipulation function used to insert a blank space in a continuous string of non blank characters. This routine is used when reading in memory address locations (see Appendix A for description of memory addresses).

3. Input - a subroutine used to read in all the task codes as character strings and to insert blank spaces between each digit of the task code.

4. Combine - a string manipulation function used to concatenate upto four strings that contain one character each. This is used when reading in the task codes that consist of four digits. Each digit is treated as a character.

5. Hex - an arithmetic function used to convert decimal numbers to hexadecimal numbers.

6. Decimal - an arithmetic function used to convert hexadecimal numbers to decimal numbers.

Some intermediate files are created during the process of the functional code generation and task code generation. These files are simply used to store intermediate results and hence may be deleted after the task is performed. There were mainly two instances at which intermediate results were stored in intermediate files:

1. When a lot of string manipulations were involved, the only possible way to do the string manipulations was to write the intermediate results to a file, and then read this intermediate file and do the desired manipulations as the strings are read in.

2. To cleanup the final output by removing extra blank spaces. The easiest way to do this is to write the results to an intermediate file and then read this file again and rewrite the results to the final file. When data is read in a second time the blank spaces are ignored. The result is a neat output.

The documentation in each of the programs lists the names of the intermediate files created. The final results are stored under the filenames provided by the user.

Table 3.2: Keywords and their attributes

Keyword	Field1	Field2	Field3	Field4	Field5
Operation_name	operation name				
Operation_number	unique operation number				
Physical_condition	name of input device	value 0 or 1	internal control relay no.		
Function	Function name	op1	op2	op3	op4 [a]
Function_Condition_1	name of input device	value 0 or 1	internal control relay no.		
Function_Condition_2	name of input device	value 0 or 1	internal control relay no.		
Action	output device or internal switch	value 0 or 1	internal device number		

[a] Where op stands for operand.

Table 3.3: Operation description format

Operation_name
Operation_number
Physical_condition
Physical_condition
.
.
.
Function
Function_condition_1 (for multiple input functions only)
Function_condition_2 (for multiple input functions only)
.
.
.
Action
Action
.
.
.

Table 3.4: Device, instruction, and function symbols

Device, Function or Instruction	Symbol
Input device	X
Output device	Y
Control relay	C
Master Control relay	MCR
Unconditional End	END
Conditional End	ENDC
Addition function	ADD
Subtraction function	SUB
Divide function	DIV
Multiply function	MULT
Square root function	SQRT
Compare function	CMP
Move Word function	MOVW
Binary to Decimal Conversion function	CBD
Decimal to Binary Conversion function	CDB
Word And function	WAND
Bit Clear function	BITC
Bit Pick and Test function	BITP
Bit Set function	BITS
Load Data Constant function	LDC
Word Rotate function	WROT
Word OR function	WOR
Word exclusive OR function	WXOR
Counter function	CTR
Timer function	TMR
Up/down Counter function	UDC
Move Word from Table function	MWFT
Move word to Table function	MWTT
Shift Bit Register	SHRB
Shift Word Register	SHRW

4 RESULTS

To verify the functioning of the framework presented in this research, the user interface module was used to create the functional code for the TI-530 PLC. The error checking module and the task code module, being system specific, were developed for this PLC. A control system was developed for a drilling station where a part is positioned, clamped, drilled and finally indexed out on a conveyor. The user interface module was used to develop the functional code for this system. This functional code was then analyzed for syntax errors and translated into the task codes for the TI-530 PLC. The system developed is discussed in the following section.

4.1 A Complete Example

The example discussed in this section is taken from the TI-530 PLC programming manual [21].

A workpiece is to be indexed automatically into a drilling station. The workpiece is clamped and drilled in the station before being indexed out on a conveyor. Figure 4.1 shows the layout of the drilling machine where:

1LS represents a sensor to indicate if the drill is in home position.

2LS represents a sensor to indicate if the drill has reached the workpiece.

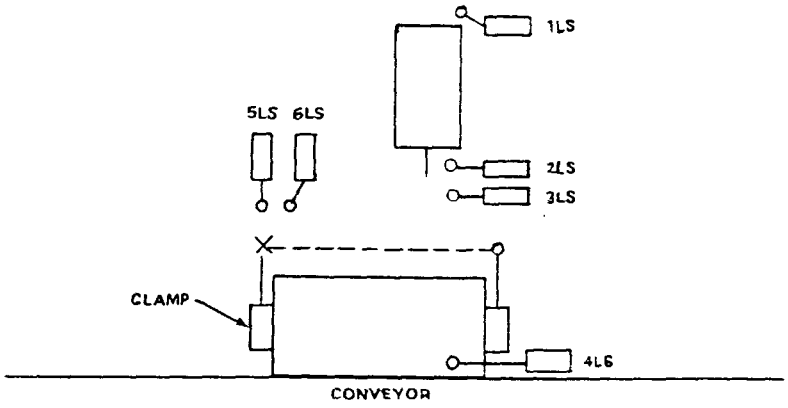


Figure 4.1: Layout of drilling machine

3LS represents a sensor to indicate if maximum drill depth is reached.

4LS represents a sensor to indicate if the workpiece is in position.

5LS represents a sensor to indicate if the workpiece is clamped.

6LS represents a sensor to indicate if workpiece is unclamped.

4.1.1 Explanation of the process

Operation 1: Index Conveyor - When the Auto_switch is in the auto mode (value 1), the workpiece is unclamped (6LS is closed), the drill is in home position (1LS is closed), and the work piece is not in home position (4LS is open) relay 1 is turned on, allowing the conveyor to index a new workpiece into the drilling station.

Operation 2: Clamp - When the auto_switch is in auto mode, the workpiece is in position (4LS is closed), the drill is not in home position (1LS is open), and drill time has elapsed (control relay 2 is on) relay 2 and relay 6 are turned on to clamp the workpiece.

Operation 3: Clamp - As long as the drill time has not elapsed, drill is in home position (1LS is closed), the workpiece is in position (4LS is closed), and work piece is not clamped (5LS is open) relay 2 and relay 6 are to stay on, that is, the work piece is to be clamped.

Operation 4: Drill down - When the workpiece is clamped (5LS is closed), drill time has not elapsed (control relay 2 is off), the auto switch is in auto mode (value 1), and maximum drill depth has not been reached (3LS is off) relay 3 is turned on to move the drill to the workpiece.

Operation 5: Start drill - When the workpiece is clamped (5LS is closed) and the drill reaches the workpiece (2LS is closed) relay 4 is turned on to start the drill.

Operation 6: Dwell timer - When the maximum drill depth is reached (3LS is on) and the drill is started (relay 4 is on) the dwell timer starts timing. The timer is reset when the drill reaches home position.

Operation 7: Drill up - When the drill time is elapsed (control relay 2 is on) and the drill is not already in home position (1LS is off) relay 5 and relay 3 are turned on to move drill back to home position.

Operation 8: Unclamp - When the drill starts its upward motion (relay 5 is on)

and maximum drill depth has not been reached (3LS is open) relay 6 is turned on to unclamp the work piece.

4.1.2 Process description used by the user interface module

The text file containing the description of the system according to the required format is shown in Table 4.1.

Table 4.1: System specification

Operation_name	index_conveyor		
Operation_number	1		
Physical_condition	Auto_switch	1	
Physical_condition	6LS	1	
Physical_condition	1LS	1	
Physical_condition	4LS	0	
Action	Relay_1	1	
Operation_name	clamp		
Operation_number	2		
Physical_condition	Auto_switch	1	
Physical_condition	Control_relay	1	2
Physical_condition	1LS	0	
Physical_condition	4LS	1	
Action	Relay_2	1	
Operation_name	clamp		
Operation_number	3		
Physical_condition	Control_relay	0	2
Physical_condition	1LS	1	
Physical_condition	5LS	0	
Physical_condition	4LS	1	
Action	relay_2	1	
Operation_name	clamp		
Operation_number	4		
Physical_condition	Auto_switch	1	

Table 4.1 (Continued)

Physical_condition	Control_relay	1	2
Physical_condition	1LS	0	
Physical_condition	4LS	1	
Action	Relay_6	1	
Operation_name	clamp		
Operation_number	5		
Physical_condition	Control_relay	0	2
Physical_condition	1LS	1	
Physical_condition	5LS	0	
Physical_condition	4LS	1	
Action	relay_6	1	
Operation_name	drill_down		
Operation_number	6		
Physical_condition	5LS	1	
Physical_condition	control_relay	0	2
Physical_condition	auto_switch	1	
Physical_condition	3LS	0	
Action	Relay_3	1	
Operation_name	Start_drill		
Operation_number	7		
Physical_condition	5LS	1	
Physical_condition	2LS	1	
Action	Relay_4	1	
Operation_name	Dwell_timer		
Operation_number	8		
Physical_condition	3LS	1	
Physical_condition	Relay_4	1	
Function	Tmr_S	1.9	
Function_condition_1	1LS	0	
Action	Control_relay	1	2
Operation_name	Drill_up		
Operation_number	9		

Table 4.1 (Continued)

Physical_condition	1LS	0	
Physical_condition	Control_relay	1	2
Action	Relay_5	1	
Operation_name	Drill_up		
Operation_number	10		
Physical_condition	1LS	0	
Physical_condition	Control_relay	1	2
Action	Relay_3	1	
Operation_name	Unclamp		
Operation_number	11		
Physical_condition	Relay_5	1	
Physical_condition	3LS	0	
Action	Relay_6	1	

4.1.3 Intermediate functional code

The following intermediate functional code specification corresponds to the example system specification.

{ { AND X 1 X 2 X 3 { NOT X 4 }} Y 1 }

{ OR { AND X 1 C 2 { NOT X 3 } X 4 } { AND { NOT C 2 } X 3

{ NOT X 5} X 4 } Y 2 }

{ OR { AND X 5 { NOT C 2 } X 1 { NOT X 6 } { AND { NOT X 3 }

C 2 } Y 3 }

{ { AND X 5 X 7 } Y 4 }

{ { AND X 6 Y 4 { TMR_S 1 { NOT X 3 } 1.9 } } C 2 }

{ { AND { NOT X 3 } C 2 } Y 5 }

{ OR { AND C 2 { NOT X 3 } X 1 X 4 } { AND { NOT C 2 } X 3

NOT X 5 } X 4 } { AND Y 5 { NOT X 6 } } Y 6 }

4.1.4 Wiring instruction

The wiring instruction for the example specification is given in Table 4.2.

Table 4.2: Wiring instruction for example system specification

Input/Output Module	Input/Output Device
X 1	Auto_switch
X 2	6LS
X 3	1LS
X 4	4LS
X 5	5LS
X 6	3LS
X 7	2LS
Y 1	Relay_1
Y 2	Relay_2
Y 3	Relay_3
Y 4	Relay_4
Y 5	Relay_5
Y 6	Relay_6

4.1.5 Task codes

The task codes for the example specification are:

1E 0000 8501 8902 8903 8B04 9901 8501 8802 8B03 8904 8602 8903 8B05 8904 9400

1E 000E 9902 8505 8A02 8901 8B06 8703 8802 9400 9903 8505 8907 9904 8506 B904

1E 001C 8703 A001 0013 9802 8703 8802 9905 8402 8B03 8901 8904 8602
8903 8B05
IE 002A B505 8B06 9400 9906

4.1.6 Ladder Logic Diagram

The ladder logic diagram corresponding to the above system specification is shown in Figure 4.2.

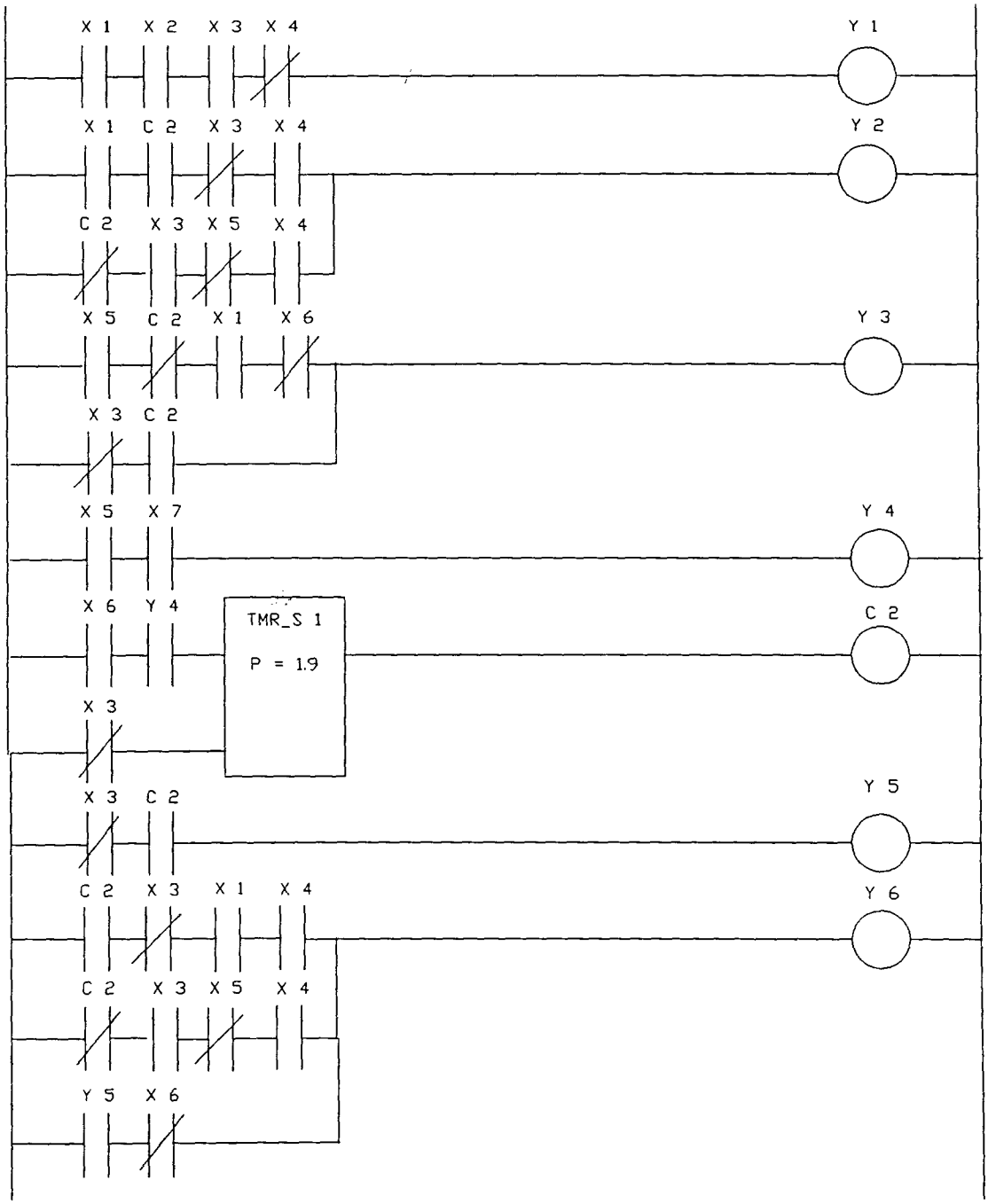


Figure 4.2: Ladder Logic Diagram for example system specification

5 CONCLUSION

The initial framework for automating the generation of control software for the TI-530 PLC was developed in this research. The system is expert system based and consists of six modules. The user defines the system to be controlled in a specification file. This specification is entered in an established format in terms of operations that consist of conditions and actions. The operations are designed such that when the conditions of the operation are true the actions are executed.

The system specification is converted to the intermediate functional code specification by the user interface module. This functional code increases the software portability of the PLCs. A wiring instruction that describes the various connections between the input and output modules of the PLC to the corresponding input and output devices is also generated.

The functional code goes through a series of tests in the error checking module to see if it violates any of the programming rules of the PLC for which the task codes must be developed. The functional code, being in standard form can easily be translated into the task code for the TI-530 PLC by the task code module. This task code module is bidirectional and hence it can also convert the given task code specification into its corresponding functional code specification. The task codes thus generated can easily be downloaded through serial communication to the PLC

for execution. On connecting the programming unit the corresponding Ladder Logic Diagram can be seen.

5.1 Future Research

PLCs have evolved a great deal since their birth in 1969. Several features such as timers, counters, arithmetic, data transfer, matrix bit-level operations have been added bringing the PLCs closer to the mini-computer. Some PLCs have been enhanced to the extent of including built-in subroutines and library functions that can be called from the user logic program as a subroutine. With PLCs having such advanced features and experiencing ongoing advancements, it can be confidently said that PLCs are here to stay. The major drawback of PLCs available in the market today is the lack of standardization of PLC programming languages and hence lack of communication between PLCs. This research presents a framework to improve the communication between PLCs by introducing an intermediate control logic representation, the functional code. This functional code can easily be translated into the machine code of any desired PLC.

The portability of the PLC software can be implemented by enhancing the code generator developed in this research. New error checking and task code modules must be developed for other PLCs since PLCs differ in their programming styles and their internal machine representation of the control logic. The functional code developed can be entered into the error checking and task code modules for the desired PLC, thus generating the machine code representation and eventually the ladder logic diagram for the given control logic.

A logic simplification module may be added to the framework presented for

efficiency in memory usage and program execution and better understandability of the control logic. The user given system specification may be simplified before it is translated into the functional code. One place where some type of simplification may be done is in the Or operations. The existing system simply combines the various And operations that have the same actions. This results in various physical conditions being listed more than once. Thus if this operation is simplified it would not only result in a clearer and easier to understand specification, but will also result in utilizing memory more efficiently. If such an approach is adapted changes may have to be made in the task code module to handle the different types of Or operations.

A module for control logic analysis may be added into the framework to evaluate some properties of a given system specification such as, conditions of deadlock, infinite loop, unsafe states, etc. This logic analysis may also be done before the system specification is translated into the functional code.

A process simulator may also be included to test the control programs before building the physical system. Simulating the system would help highlight any problems that might occur when implementing the actual system.

6 APPENDIX A USER MANUAL FOR THE USER INTERFACE MODULE

The user interface module requires a description of the process to be controlled as input. This description needs to be in a specific format. The following rules need to be followed when creating the control system description:

1. A process to be controlled must be broken down into control steps called operations.

2. An operation is identified by a name and a unique number. Keywords and the pattern specified in Table 6.1 must be used to describe an operation. The keywords 'operation_name' and 'operation_number' are used to specify the name and number of the operation respectively. The physical conditions, functions and actions of an operation are specified by the key words 'physical_condition', 'function', and 'action' respectively. The input conditions of a multiple input function are specified using the keywords 'function_condition_1' and 'function_condition_2'.

3. All operation and device names must be alphanumeric without any blank spaces. For example: Limit_switch_1, Advance_conveyor, etc.

4. All device names referring to different devices must be unique. Sequential numbering may be used to distinguish between devices of the same type. For example, if two solenoids are used in a process the two solenoids may be named as 'solenoid_1' and 'solenoid_2'.

5. The operation description must follow the order shown in Table 6.2.

6. Some common PLC internal instructions, internal devices, and functions along with their operands are shown in Tables 6.3, 6.4, and 6.5. These instructions, devices and functions must be referred to by the given abbreviations.

7. All internal devices and instructions must be numbered except the conditional and unconditional end instructions.

8. A function may not be the first condition of an operation. Hence an operation must have at least one physical_condition specified before a function can be specified.

9. An operation having a combination of multiple and single input functions requires that the multiple input function be specified before the single input function is specified.

10. All functions must have the given number of operands in the given order.

11. Multiple input functions may have as many as two types of input conditions and a maximum of two of each type of input condition.

Table 6.1: Keywords and their attributes used in the system description

Keyword	Field1	Field2	Field3	Field4	Field5
Operation_name	operation name				
Operation_number	unique operation number				
Physical_condition	name of input device	value 0 or 1	internal control relay no.		
Function	Function name	op1	op2	op3	op4 [a]
Function_Condition_1	name of input device	value 0 or 1	internal control relay no.		
Function_Condition_2	name of input device	value 0 or 1	internal control relay no.		
Action	output device or internal switch	value 0 or 1	internal device number		

[a] Where op stands for operand.

Table 6.2: Operation description format

Operation_name
Operation_number
Physical_condition
Physical_condition
.
.
.
Function
Function_condition_1 (for multiple input functions only)
Function_condition_2 (for multiple input functions only)
.
.
.
Action
Action
.
.
.

Table 6.3: Common PLC internal devices and instructions

Instruction or Device	Symbol
Control Relay	Control_relay n [a]
Master Control Relay	MCR n
End Master Control Relay	MCRE n
Unconditional End	END
Conditional End	ENDC

[a] Where n refers to the instruction number.

Table 6.4: Single input functions and their operands

Function	Function name	Operands
Add	ADD	Op1: Memory location of first value to be operated on Op2: Memory location of second value to be operated on Op3: Memory location for storage of result
Subtract	SUB	Op1: Memory location of first value to be operated on Op2: Memory location of second value to be operated on Op3: Memory location for storage of result
Divide	DIV	Op1: Memory location of dividend Op2: Memory location of divisor Op3: Memory location of quotient and remainder
Multiply	MULT	Op1: Memory location of multiplicand Op2: Memory location of multiplier Op3: Memory location of product
Square root	SQRT	Op1: Memory location of value to be operated on Op2: Memory location for storage of result
Compare	CMP	Op1: Memory location of value to be compared with Op2: Memory location of value being compared with Op3: Device indicating Op1 is less than Op2 Op4: Device indicating Op1 is greater than Op2
Move Word	MOVW	Op1: Starting address of words to be moved Op2: Starting destination address for words to be moved to

Table 6.4 (Continued)

Function	Function name	Operands
Load Data Constant	LDC	Op1: Memory location for data constant Op2: Data constant to loaded
Move Image Register to word	MIRW	Op1: Device type and number Op2: Destination word address Op3: Number of bits to be moved
Move word to Image Register	MWIR	Op1: Source word address Op2: Device type and number Op3: Number of bits to be moved
Convert Binary to Decimal	CBD	Op1: Source memory address Op2: Destination memory address
Convert Decimal to Binary	CDB	Op1: Source Memory address Op2: Destination memory address Op3: Number of bits to be converted
Word And	WAND	Op1: Memory location of first word Op2: Memory location of second word Op3: Memory location where the word is to be stored
Word Or	WOR	Op1: Memory location of one word to be acted upon Op2: Memory location of second word to be acted upon Op3: Memory location for the storage of result
Word Rotate	WROT	Op1: Memory location of word to be rotated Op2: Number of rotations
Word Exclusive Or	WXOR	Op1: Memory location of one word to be acted upon Op2: Memory location of second word to be acted upon Op3: Memory location for the storage of result

Table 6.4 (Continued)

Function	Function name	Operands
Bit clear	BITC	Op1: Memory location of word to be acted upon Op2: Number of bits to be cleared
Bit pick	BITP	Op1: Memory location of word to be acted upon Op2: Number of bits to be tested
Bit set	BITS	Op1: Memory location of word to be acted upon Op2: Number of bits to be set
One PC scan (One/Shot)	O/S	

The system is user friendly. A menu driven approach has been taken for ease of use. The user is given five options:

1. Convert a system specification to the intermediate functional code specification.

2. Perform error checking on a given intermediate functional code specification.

3. Convert a given functional code to the corresponding task code.

4. Convert a given task code specification to the corresponding functional code.

5. Exit

For option 1 it is required that the user have the system specification in an ASCII file. The user is asked for the name of the file that contains the control system description and the name of the output file that is used to store the intermediate functional code. For option 2 the user is asked for the names of files that contain the functional code and the one to which all error messages are written to. Under options 3 and 4 the names of the files with the functional code and the task code are prompted for. Option 5 enables the user to exit the system when the user is done executing the program. This menu is displayed at the end of every task performed.

Table 6.5: Mutiple input functions and their operands

Function	Function name and condition	Operands
Counter	CTR	Op1: Count upper limit
	Function_condition_1	Enable condition
Millisecond Timer	TMR_MS	Op1: Preset time
	Function_condition_1	Enable/Reset condition
Seconds timer	TMR_S	Op1: Preset time
	Function_condition_1	Enable/Reset condition
Move word from table	MWFT	Op1: Location of table address pointer for next word to be moved
		Op2: Destination address
		Op3: Number of words to be moved
		Op4: Starting address of table
	Function_condition_1	Reset condition
Move word to table	MWTT	Op1: Source address of word
		Op2: Location of table address pointer
		Op3: Number of words to be moved
		Op4: Starting address of table
	Function_condition_1	Reset condition
Bit Shift Register	SHRB	Op1: Starting address of shift register
		Op2: Number of bits in the shift register
	Function_condition_1	Data input condition
	Function_condition_2	Reset/enable condition
Word Shift Register	SHRW	Op1: Memory location of the word to be placed in the shift register
		Op2: Memory location of start of shift register
		Op3: Number of words in shift register
	Function_condition_1	Enable condition
	Function_condition_2	Reset condition
Up/Down	UDC	Op1: Preset value of counter
		Op2: Device to be energized when count is zero
	Function_condition_1	Down condition
	Function_condition_2	Reset/Enable condition

7 APPENDIX B USER MANUAL FOR THE TI-530 PLC

When creating the file with the specification for the system to be controlled using the TI-530 PLC it is important to follow certain syntax rules. These rules have been specified in the TI-530 PLC manual. They have been summarized here. The error checking module checks for violations of any of these rules.

1. The internal instructions, internal devices, and functions available on the TI-530 PLC are listed in Tables 7.1 and 7.2. These must be referred to by the symbols specified. In the case of functions the operands must be used in the order indicated. The types of memory addresses to be used as various operands in also shown. Table 7.3 shows the different addresses and the address ranges available on the TI-530 PLC.

2. The memory addresses shown in Table 7.3 are straightforward except the DCP address. This address indicates the drum number and the step number. The address ranges from 101 to 3016. The two least significant digits represent the step number and the remaining digit(s) represent the drum number. For example, address location DCP1020 represents drum number 10 and step number 19 (zero based).

3. Each operation may have as many as 11 physical conditions.

4. Each operation may have as many as 7 actions.

5. Each operation must have at least 1 action associated with it.

6. An operation may have as many as three single input functions.

7. If an operation has only one single input function associated with it then the operation may have as many as 8 physical conditions.

8. If an operation has two single input functions associated with it then the operation may have as many as 5 physical conditions.

9. If an operation has three single input functions associated with it then the operation may have as many as 2 physical conditions.

10. An operation may have only one multiple input function associated with it.

11. An operation that has a multiple input function may not have more than two physical conditions specified before it.

12. An operation that has one multiple input function, this function may be followed by up to two single input functions or six physical conditions in series.

13. An operation with one multiple input function and one single input function may not have more than five physical conditions , two preceding the multiple input function and three following it. The three physical conditions can precede or follow the single input function in any combination.

14. An operation with a multiple input function and two single input functions may have upto two physical conditions, but these physical conditions may only precede the multiple input function.

Table 7.1: TI-530 PLC internal devices and instructions

Instruction or Device	PLC Coil Symbol
Jump	JMP n [a]
End Jump	JMPE n
Control Relay	Control_relay n
Master Control Relay	MCR n
Master Control Relay End	MCRE n
Unconditional End	END
Conditional End	ENDC

[a]Where n refers to the instruction reference number.

Table 7.2: TI-530 PLC functions

Functions	Operands
Single Input Functions	
ADD	Memory Location of first value
(Add)	(V, WX, WY)
	Memory Location of second value
	(V, WX, WY)
	Memory location for storage of result
	(V, WX, WY)
SUB	Memory Location of first value
(Subtract)	(V, WX, WY)
	Memory Location of second value
	(V, WX, WY)
	Memory location for storage of result
	(V, WX, WY)
DIV	Memory Location of dividend
(Divide)	(V, WX, WY)
	Memory Location of divisor
	(V, WX, WY)
	Memory location of quotient and remainder
	(V, WY)
MULT	Memory Location of multiplicand
(Multiply)	(V, WX, WY)
	Memory Location of multiplier
	(V, WX, WY)
	Memory location of product
	(V, WY)
SQRT	Memory Location of value
(Square root)	(V, WX, WY)
	Memory Location for storage of result
	(V, WY)
CMP	A: Memory Location of value to be
(Compare)	compared with
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	B: Memory Location of value being
	compared
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Output device or control relay indicating

Table 7.2 (Continued)

Functions	Operands
	A is less than B
	(C, Y)
	Output device or control relay indicating A is greater than B
	(C, Y)
MOVW	Starting address of words to be moved
(Move Word)	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Starting destination address for words to
	(TCP, DSP, DCP, V, WY)
	be moved to
	Number of words to be moved
	(1-256)
CBD	Source Memory address
(Convert Binary to Decimal)	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Destination Memory address
	(V, WY)
CDB	Source Memory address
(Convert Decimal to Binary)	(V, WX, WY)
	Destination Memory address
	(TCP, DSP, DCP, V, WY)
	Number of digits to be converted
	(1-4)
WAND	Memory location of first word
(Word And)	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Memory location of second word
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Memory location where the word is to be stored
	(TCP, DSP, DCP, V, WY)
BITC	Memory location of word to be operated on
(Bit Clear)	(V, WY)
	Number of bit to be cleared
	(1-16)
BITP	Memory location of word to be operated on
(Bit Pick)	(V, WX, WY)

Table 7.2 (Continued)

Functions	Operands
	Number of bit to be tested
	(1-16)
BITS	Memory location of word to be operated on
(Bit Set)	(V, WY)
	Number of bit to be set
	(1-16)
LDC	Memory location for data constant
(Load Data Constant)	(TCP, DSP, DCP, V, WY)
	Data constant to be loaded
	(0-32767)
MIRW	Input or Output device or
(Move Image Register to Word)	internal control relay
	(X, Y, C)
	Destination word address
	(TCP, DSP, DCP, V, WY)
	Number of bits to be moved
	(1-16)
MWIR	Source word address
(Move Word to Image Register)	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Input or Output device or
	internal control relay
	(X, Y, C)
	Number of bits to be moved
	(1-16)
WROT	Memory location of word to
(Word Rotate Right)	be rotated
	(V, WY)
	Number of times the 4-Bit segments are to
	be rotated
	(1-3)
WOR	Memory location of one word to be
(Word Or)	acted upon
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Memory location of second word to be
	acted upon
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)

Table 7.2 (Continued)

Functions	Operands
	Memory location for the storage of result (TCP, DSP, DCP, V, WY)
WXOR (Word Exclusive Or)	Memory location of one word to be acted upon (TCC, TCP, DSC, DSP, V, WX, WY) Memory location of second word to be acted upon (TCC, TCP, DSC, DSP, V, WX, WY) Memory location for the storage of result (TCP, DSP, DCP, V, WY)
O/S (One/Shot)	
Multiple Input Functions	
CTR (Counter)	Count upper limit (1-32,767)
Function_condition_1	Enable condition
TMR_MS (Timer Millisec)	Preset time which timer times down from
Function_condition_1	Enable/Reset condition
TMR_S (Timer .1 Sec)	Preset time which timer times down from
Function_condition_1	Enable/Reset condition
MWFT (Move Word From Table)	Location of table address pointer for next word to be moved (V) Destination address for word to be moved (V) Number of words to be moved (1-256) Starting address of table (V)
Function_condition_1	Reset condition
MWTT (Move Word To Table)	Source address of word to be moved (V) Location of table address pointer (V)

Table 7.2 (Continued)

Functions	Operands
	Number of words to be moved
	(1-256)
	Starting address of table
	(V)
Function_condition_1	Reset condition
SHRB	Starting address of shift register
(Bit Shift Register)	(C, Y)
	Number of bits in shift register
	(1-16)
Function_condition_1	Data input
Function_condition_2	Reset/enable condition
SHRW	Memory location of word to be placed
(Word Shift Register)	in shift register
	(TCC, TCP, DSC, DSP, DCP, V, WX, WY)
	Memory location of start of shift
	register
	(V)
	Number of words in shift register
	(1-16)
Function_condition_1	Enable condition
Function_condition_2	Reset condition
UDC	Preset value of the maximum value to
(Up/Down Counter)	which the counter will count
	(1-32767)
	The output device or internal control
	relay to be energized when the
	count is zero
	(C, Y)
Function_condition_1	Down condition
Function_condition_2	Reset/Enable condition

Table 7.3: Memory addresses used as function operands

Address type	Address range
X	1 - 1023
Y	1 - 1023
C	1 - 511
V	1 - 1024
WX	1 - 1023
WY	1 - 1023
TCC	1 - 255
TCP	1 - 255
DSC	1 - 30
DSP	1 - 30
DCP	101 - 3016

8 APPENDIX C TASK CODES FOR THE TI-530 PLC

The task codes of the TI-530 PLC are four digit hexadecimal numbers. There is a different task code for each of the different types of devices, instructions, functions, addresses. The different task codes for the devices, instructions, functions, and addresses are shown in Tables 8.1 and 8.2.

In the case of instructions, devices, and functions the first two most significant digits represent the type of device, instruction, or function, the remaining two digits are the hexadecimal representation of the number of the device, instruction, or function. In the case of memory address location the most significant digit indicates the starting address location and the remaining three digits indicated the offset.

Table 8.1: Task codes for TI-530 PLC

Description	Symbol	Task code
First Physical Condition	X	85nn [a]
(Device number less than 256)	Not X	87nn
	Y	B5nn
	Not Y	B7nn
	C	84nn
	Not C	86nn
First Physical Condition	X	E5xx [b]
(Device number between 256 and 511)	Not X	E7xx
	Y	C5xx
	Not Y	C7xx
	C	E4xx
	Not C	E6xx
First Physical Condition	X	B4xx
(Device number between 512 and 767)	Not X	B6xx
	Y	A5xx
	Not Y	A7xx
First Physical Condition	X	F4xx
(Device number between 768 and 1023)	Not X	F6xx
	Y	C4xx
	Not Y	C6xx
Following Physical Condition	X	89nn
(Device number less than 256)	Not X	8Bnn
	Y	B9nn
	Not Y	BBnn
	C	88nn
	Not C	8Ann
Following Physical Condition	X	E9xx
(Device numbe between 256 and 511)	Not X	EBxx

[a] Where nn indicates the device or function number in hexadecimal.

[b] Where xx indicates the last two digits of device number in hexadecimal.

Table 8.1 (Continued)

Description	Symbol	Task code
	Y	C9xx
	Not Y	CBxx
	C	E8xx
	Not C	EAxx
Following Physical Condition	X	B8xx
(Device numbe between 512 and 767)	Not X	BAxx
	Y	A9xx
	Not Y	ABxx
Following Physical Condition	X	D8xx
(Device numbe between 768 and 1023)	Not X	DAxx
	Y	C8xx
	Not Y	CAxx
Single Or Physical condition	X	8Dnn
(Device number less than 256)	Not X	8Fnn
	Y	BDnn
	Not Y	BFnn
	C	8Cnn
	Not C	8Enn
Single Or Physical condition	X	EDnn
(Device number between 256 and 511)	Not X	EFxx
	Y	CDxx
	Not Y	CFxx
	C	ECxx
	Not C	EExx
Single Or Physical condition	X	BCnn
(Device number between 512 and 767)	Not X	BExx
	Y	A1xx
	Not Y	A3xx
Single Or Physical condition	X	DCnn
(Device number between 768 and 1023)	Not X	DExx
	Y	C0xx
	Not Y	C2xx
Action	Y	99nn
(Device number less than 256)	Not Y	9Bnn
	C	98nn

Table 8.1 (Continued)

Description	Symbol	Task code
	Not C	9Ann
	JMP	92nn
	JMPE	96nn
	MCR	9Cnn
	MCRE	9Enn
	END	FF01
	ENDC	FE01
Action	Y	F9nn
(Device number between 256 and 511)	Not Y	FBxx
	C	E0xx
	Not C	E2xx
Action	Y	ADnn
(Device number between 512 and 767)	Not Y	AFxx
Action	Y	CCnn
(Device number between 768 and 1023)	Not Y	CExx
Function	ADD	D0nn
	SUB	D1nn
	CMP	D6nn
	DIV	D3nn
	MULT	D2nn
	SQRT	D4nn
	BITC	F2nn
	BITP	F0nn
	BITS	F1nn
	SHRB	DBnn
	CBD	FDnn
	CDB	FCnn
	WAND	FAnn
	WOR	F8nn
	WROT	E3nn
	WXOR	E1nn
	SHRW	D9nn
	LDC	F3nn
	MIRW	D7nn
	MWIR	D5nn

Table 8.1 (Continued)

Description	Symbol	Task code
	MOVW	F5nn
	MWFT	C3nn
	MWTT	C1nn
	CTR	A8nn
	TMR_S	A0nn
	TMR_MS	ACnn
	UDC	A4nn
	O/S	F7nn
Or Operation with more than 1 Physical Condition		9400

Table 8.2: Memory areas and their corresponding task codes

Memory area	Task code
X1	4001
.	.
.	.
X1023	43FF
Y1	6001
.	.
.	.
Y1023	63FF
C1	0001
.	.
.	.
C511	01FF
V1	0000 (Zero based)
.	.
.	.
V1024	03FF
WX1	6001
.	.
.	.
WX1023	63FF
WY1	7001
.	.
.	.
WY1023	73FF
TCP1	4001
.	.
.	.
TCP128	4080
TCP129	4101
.	.
.	.
TCP255	417F

Table 8.2 (Continued)

Memory area	Task code
TCC1	4081
.	.
.	.
TCC128	4100
TCC129	4181
.	.
.	.
TCP255	41FF
DSP1	5001
.	.
.	.
DSP30	501E
DSC1	501F
.	.
.	.
DSC30	503C
DCP101	1010
.	.
.	.
DCP3016	11EF

9 BIBLIOGRAPHY

[1] F. Aldana, J. Peire, C. M. Penalver and J. Uceda; "Computer Aided Generation of Microprocessor Software for Controlling Static Power Converters"; Proceedings of the Third IFAC/IFIP Symposium on Software Computer Control 1982; Pergamon Press: Oxford, England; 249-253.

[2] H. Atabakhche, D. S. Barbalho, R. Valette, and M. Courvoisier; "From Petri Net Based PLCs to Knowledge Based Control"; Proceedings IECON'86 1986 International Conference on Industrial Electronics, Control and Instrumentation; Institute of Electrical and Electronics Engineers: York, NY; 1986; Vol 2; 817-822.

[3] A. D. Baker, t. L. Johnson, D. I. Kerpelman, and H. A. Sutherland; "GRAFCET and SFC as Factory Automation Standards Advantages and Limitations"; 1987 American Control Conference; American Automatic Control Council: Green Valley, AZ; 1987; Vol 3; 1725-1730.

[4] G. Balbo, G. Chiola, G. Franceschinis, and G. M. Roet; "Generalized Stochastic Petri Nets for the Performance Evaluation of FMS"; Proceedings 1987 IEEE Conference on Robotics and Automation; Computer Society Press of IEEE: Washington, D.C.; 1987; Vol 2; 1013-1018.

[5] G. Bruno and G. Marchetto; "Process Translatable Petri Nets for the Rapid Prototyping of Process Control Systems"; IEEE Transactions on Software Engineering; IEEE Computer Press Society: Washington, D.C.; 1986; Vol SE-12, No. 2; 346-357.

[6] R. A. Chard; Software Concepts in Process Control; NCC Publications: Manchester, England; 1983; 119-150.

[7] M. Courvoisier, R. Valette, J. M. Bigou, and P. Esteban; "A Programmable Logic Controller Based on a High Level Specification Tool"; International Conference on Industrial Electronics, Control, and Instrumentation: IEEE Proceedings of the IECON'83; Institute of Electrical and Electronics Engineers: New York, NY; 1983; 174-179.

[8] J. C. Gentina and D. Corbeel; "Coloured Adaptive Structured Petri-Net: A Tool for the Automatic Synthesis of Hierarchical Control of Flexible Manufacturing Systems (F.M.S.)"; IEEE 1987 International Conference on Robotics and Automation; Computer Society Press of the IEEE: Washington, D.C.; 1987; Vol 2; 1166-1173.

[9] K. W. Golf; "Artificial Intelligence in Process Control"; Mechanical Engineering; American Society of Mechanical Engineers: New York, NY; Oct 1985; Vol 107/No. 10; 53-57.

[10] C. Gomez, J. P. Quadrat, A. Sulem, G. L. Blankenship, P. Kumar, A. LaVigna, D. C. MacEnany, K. Paul, and I. Yan; "An Expert System for Control and Signal Processing with Automatic FORTRAN Code Generation"; Proceedings of 23rd Conference on Decision and Control; Institute of Electrical and Electronics Engineers: New York, NY; 1984; 716-723.

[11] S. J. Grant; "Programming Languages and Programmable Controllers"; Proceedings of the First Annual Control Engineering Conference; Control Engineering: Barrington, IL; 1982; 95-97.

[12] M. Groover; Automation, Production Systems, and Computer Integrated Manufacturing; Prentice-Hall: Englewood Cliffs, NJ; 1987; 281-424.

[13] M. Groover; Automation, Production Systems, and Computer Integrated Manufacturing; Prentice-Hall: Englewood Cliffs, NJ; 1980; 642-669.

[14] H. Hanselmann and A. Schwarte; "Generation of Fast Target Processor Code from High Level Controller Descriptions"; IFAC 10th Triennial World Congress; Pergamon Press: Oxford, England; 1987; Vol 4; 93-98.

[15] M. Kamath and N. Viswanadham; "Applications of Petri Net Based Models in the Modelling and Analysis of Flexible Manufacturing Systems"; IEEE International Conference on Robotics and Automation; IEEE Computer Press Society: Washington, D.C.; 1986; Vol 1; 312-317.

[16] M. S. King, S. L. Brooks, and R. M. Schaefer; "Knowledge Based Systems"; Mechanical Engineering; American Society of Mechanical Engineers: New York, NY; Oct 1985; Vol 107/No. 10; 58-61.

[17] Komoda, Kero, and Kubo; "An Autonomous, Decentralized Control System for Factory Automation", Computer; IEEE Computer Society Press: Washington, D.C.; Dec 1984; Vol 17, No. 12; 73-83.

[18] B. H. Krogh and G. Ekberg; "Automatic Programming of Controllers for Discrete Manufacturing Processes", IFAC 10th Triennial World Congress; Pergamon Press: Oxford, England; 1987; Vol 4; 161-165.

[19] B. H. Krogh, R. Wilson, and D. Pathak; "Automatic Generation of Control Programs for Discrete Manufacturing Processes"; The Robotics Institute 1987 Annual Research Review; Carnegie Melon University: Pittsburgh, Pennsylvania; 1987; 21-31.

[20] F. LeGland and A. Gondel; "Systematic Numerical Experiments in Nonlinear Filtering with Automatic Fortran Code Generation"; Proceedings of IEEE Conference on Decision and Control, Institute of Electrical and Electronics Engineers: New York, NY; 1986; 638-642.

[21] Model 530 Programmable Controller Program Design Guide Manual No. 530-8104; Texas Instruments Incorporated; Second Edition; 1983.

[22] Y. Narahari and N. Viswanadham; "A Petri Net Approach to the Modelling and Analysis of Flexible Manufacturing Systems"; Annals of Operations Research; Baltzer: Basel, Switzerland; 1985; Vol 3-4, 449-473.

[23] Non-Intelligent Terminal Protocol Compatibility Specification; Texas Instruments Incorporated: Johnson City, Tennessee; 1981; 1-12.

[24] D. Pathak and B. H. Krogh; "Concurrent Operation Specification Language, COSL, for Low-Level Manufacturing Control"; Paper; ECE Dept; Carnegie Mellon University; 1988; 1-26.

[25] A. Sahraoui, ?H. Atabakhche, M. Courvoisier, and R. Valette; "Joining Petri Nets and Knowledge Based Systems for Monitoring Purposes"; IEEE 1987 International Conference on Robotics and Automation; Computer Society Press of the IEEE: Washington, D.C.; 1987; Vol 2; 1160-1165.

[26] Series 500 Communication Task Code Compatibility Specification; Texas Instruments Incorporated: Johnson City, Tennessee; 1987; 1-116.

[27] W. H. Simmonds; "Future Trends in Process Control"; Electronics and Power, Institute of Electrical Engineers: London, England; Sept 1987; Vol 33, No. 9; 570-572.

[28] R. J. Srodawa, R. E. Gach, and A. Glicker; "Preliminary Experience with the Automatic Generation of Production-Quality Code for the Ford/Intel 8061 Microprocessor"; IEEE Transactions on Industrial Electronics; Institute of Elec-

trical and Electronics Engineers: New York, NY; 1985; Vol IE-32, No. 4; 318-326.

[29] H. Tanaka, C. Nakajima, and M. Yoshida; "Intermediate Functional Language FCL for Improving Software Portability of Programmable Controllers"; Proceedings of the USA-JAPAN symposium on Flexible Automation; ACME: New York, NY; 1988; Vol 2; 1149-1154.

[30] Tashiro, Komoda, Tsushima, and Matsumoto; "Advanced Software for Constraint Combinational Control of Discrete Event Systems - Rule-based Control Software for Factory Automation"; IEEE 1985 COMPINT-Computer technologies; IEEE Computer Society Press; 1985; 73-83.

[31] T. Tashiro, N. Komoda, I. Tsushima, and K. Matsumoto; "Rule-Based Control Software System for Factory Automation - Its Rule Correctness Check Support And Response-Time Estimation";Software For Computer Control 1986 Selected Papers from the Fourth IFAC/IFIP Symposium, Graz, Austria; Pergamon Press: Oxford, England; 1987; 53-58.

[32] T. J. Williams, The Use of Digital Computers in Process Control; Instrument Society of America:Research Park, N.C.; 1984; 137-151.

[33] R. Wilson and B. H. Krogh; "Specification and Analysis of Control Separation Models for Discrete State Systems"; Paper; ECE Dept.; Carnegie Mellon University; 1988; 1-29.

[34] J. T. Woon; "What the Future Holds For Programmable Controller Languages"; Proceedings of the First Annual Control Engineering Conference; Control Engineering: Barrington, IL; 1982; 91-93.

[35] M. D. Zisman; "Use of Production Systems for Modeling Asynchronous, Concurrent Processes"; Pattern-Directed Inference Systems; D. A. Waterman and F. Hayes-Roth; Academic Press: New York, NY; 1978; 53-68.