Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1992

Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler

N. Meghamala
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1992-015

Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler

N. Meghamala

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler

by

N. Meghamala
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #92-015 Dec, 1992

Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler

by N. Meghamala

submitted in partial fulfillment of the requirements for

Master's degree in Systems Analysis

Miami University

Oxford, Ohio

December, 1992

Development of an Object-Oriented High_Level Language and Construction of an Associated Object-Oriented Compiler

Abstract

Computerized automation has long past been in practice, and engineers are developing ways and means of improving strategies to achieve better performance. Related research in software is concentrated on developing methods to diversify the application of computing technologies by bringing software and its field of application close together. The idea is to eventually create a software configuration that makes itself transparent to the user by mapping reality onto the computer's monitor. This paper is a report on a portion of an interdisciplinary project that attempts to apply state-of-the-art software techniques in the design of an integrated programming environment for manufacturing purposes. The portion of the project described in this paper is concerned with two aspects: designing a special purpose object-oriented high-level language for programming a classroom research laboratory for manufacturing students and faculty, and designing and implementing an object-oriented compiler to allow students to write programs using this language. The purpose of the project is to integrate manufacturing devices and cells with the existing computer network to provide a flexible manufacturing system.

1. Introduction

Manufacturing engineering is a classic example of a field faced with problems of interfacing components of a heterogeneous environment. The advent of electronic technology introduced programmable devices with resident memory to improve automation techniques. But, engineers were faced with some new problems: the lack of an efficient means of controlling the operations on devices due to the absence of an integrated environment and a common platform to monitor all devices; a highly error prone cumbersome process of writing a program in the command languages of programmable devices such as CNC Machine, Robot, etc., and the loss of control over the execution of a manufacturing process once a command language program has been loaded into the device's programmable controller.

This was the problem that confronted the classroom laboratory of the Department of Manufacturing Engineering at Miami University. The addition of computers into the manufacturing arena provided a network to integrate the components of a manufacturing cell, but did not eliminate the need to program individual devices using command languages. Furthermore, the operator was still not able to retain control over a programmable device once a program was loaded into its memory. Though the requisite hardware interface was present to network the cells, there was no software available to complement the integration. In addition, the current hardware served to interface components of a single manufacturing cell, but not entire cells. There was a need to concoct a solution (or solutions) that would solve all of the above problems, and provide a flexible computer controlled environment to conduct automated manufacturing processes.

The goal of the project described in this paper is to provide an integrated programming environment for programming individual manufacturing cells by developing a special purpose high-level programming language. Eventually, the programming environment will be enhanced to provide for integrating the manufacturing cell network as a whole. This is part of a larger project undertaken by the departments of Systems Analysis, Manufacturing Engineering, and Engineering Technology to provide a classroom Computer Integrated Manufacturing (CIM) laboratory for students, and to provide a research laboratory for the faculty to test state-of-the-art automation strategies[7]. The main idea behind developing a special purpose language is to replace the robot controller, along with its cryptic language, as the controller for the entire cell and also to be able to give control of the cell and the devices to the programmer/operator even after a command language program is loaded into the programmable controllers of these devices.

Cell Programming Language or CPL is the name of the programming language environment that we chose for use in the control of manufacturing cells. Individual cell components and their operations can be integrated by programming the cell as a single unit. Programs to do this could be written in any other existing high-level language such as BASIC or C, but the user would have to be familiar with the syntax necessary to perform low-level input and output to the hardware that provides the interface to the cell's devices. For example, the

user would have to set a particular bit on a particular hardware port to 1 to turn on a device. Instead, CPL allows the user to program the cell by using commands such as On and Off, and the CPL system takes care of the low-level programming details.

The purpose of this paper is to present the results of applying the data abstraction capabilities of object-oriented methodology to a familiar aspect of day-to-day life, *viz.*, computer aided manufacturing. The following sections present a report on the project undertaken with a detailed description of the problem to be solved and the solution offered. Section 2.0 discusses object-oriented techniques in relation to manufacturing engineering software. It also discusses the rationale for using the object-oriented paradigm in building a Flexible Manufacturing System (FMS)- a term that indicates the built-in resilience of object-oriented CIM systems. The next section describes the basic concepts of object-oriented design. Section four describes the existing CIM environment which is also the target system for the project presented in this paper. The fifth section deals with issues concerning language design and its relevance to the subject in question and defines a grammar for CPL. The description of its syntax are discussed in section six and the remaining sections explain the design of a compiler for the language and future directions for this project.

2. Manufacturing Engineering: An Object-Oriented Perspective

The object-oriented paradigm can lend itself to any problem domain in the universe. This is because it is based on the simple idea that any concept can be regarded as an abstraction, and be represented as an object which is characterized by attributes and a set of operations that can be performed on it. The growing popularity of the paradigm in more than one area of technology is seen as an indication of how much closer the propounded ideas have brought the application of computing to the other branches of science and technology, and thus to the real world.

Representing key abstractions of a domain in terms of objects allows the software user to stay in contact with the real-world that is being simulated by the software. This is the principle advantage of object-orientation. This feature is especially important in manufacturing engineering where engineers constantly operate tangible objects. The fact that a computer

operator of a manufacturing system can attach the same tangibility to data abstractions used in a computer program is of paramount importance. In assembly line product manufacturing, the cost of a small error is very high, and hence it is very crucial to the computer operator that he/she "sees" the machine or device while performing an operation on it.

In the object-oriented design paradigm, analysis and design activities are performed close together to build a model of the system in the problem domain. The system is built as an entity-relationship model where the entities transcend the problem domain right through the entire development cycle. At each stage of object-oriented development, the entities or objects directly represent the actual objects. This approach to systems development views the real-world system as a collection of objects or entities, and the paradigm functions on this essence of reality. A client-server model is used to analyze and design the system in terms of the services provided by each object and the relationships among different objects. The analysis and design phases concentrate on identifying the objects in the real-world and mapping them to programming objects. This mapping is, however, accomplished with a strict adherence to information hiding and data encapsulation principles. The specification of procedures and data structures is performed at a much later stage during the detailed design of the system. The paradigm provides a robust framework to analyze the system, and design individual objects independent of implementation details. This renders the design as a flexible yet well integrated representation of the system. It also provides additional valuable qualities such as information hiding, reusability, encapsulation, etc.

Commercially available manufacturing software until recently was based on the process-centric paradigm [3] of software engineering. This methodology has undergone a severe crisis for the past two decades. In these software systems, the end-user was totally separated from the actual system, since data abstraction was not implicit in the discipline used to build the system and in the languages used to program devices and cells. The problems in such systems are manifold, since the programmer of these machines has to deal with languages that are highly cryptic and code-like, even for specifying high-level control commands and operations. For example, the code segment used to program a cell using a robot controller, shown in figure 1.0, contains control commands and device specification commands in a single pro-

gram. This combination does not provide for data and procedure encapsulation. One can see that this highly cryptic code is not related to the problem domain, and is difficult to maintain.

```
5 LPRINT "PS 1, -2400, -1600, 800, 1570, 1390, 0"
10 LPRINT "PS 2, -2400, -1840, 260, 1795, 1165, 0"
15 LPRINT "GC"
20 LPRINT "GF 1"
25 LPRINT "PS 3, -2400, -1840, 260, 1795, 1165, 0"
30 LPRINT "PS 4, -2400, -820, 20, 1740, 1220, 0"
35 LPRINT "PS 5, 9600, -620, 1020, 1740, 1220, 0"
40 LPRINT "PS 6, -9350, -2420, 2720, 1740, 1220, 0"
45 LPRINT "PS 7, -9330, -2950, 2895, 1740, 1220, 0"
50 LPRINT "PS 8, -9460, -2950, 2895, 1740, 1220, 0"
55 LPRINT "GO"
60 LPRINT "GF 0"
65 LPRINT "PS 9,  -9460, -2950, 2895, 1740, 1220, 0"
```

Figure 1.0 BASICA program that uses robot controller code

In order to apply Object-Oriented methodology to the manufacturing domain, it is important to relate the two by identifying characteristics of the domain that can be easily incorporated into the methodology. S.Y. Nof [6], gives a detailed list of the characteristics, some of which are discussed in this paragraph, that contribute to the affiliation between the two areas. Analogies may be drawn between the data abstraction capabilities of Object-Orientation and the use of tangible machinery and combination of processes to produce concrete parts.

In manufacturing processes, data and control are combined to achieve the requisite material processing to create end products. These processes may be represented by an integrated data and process model provided by Object_Oriented Design. The distinct identification of separable manufacturing units and the minimal dependency between them is easily incorporated by the encapsulation and information hiding features. The commanility between various components can be addressed by the inheritance mechanisms present in this paradigm. Other features such as polymorphism and dynamic binding are suitable for implementing assembly line processing of similar parts. These two characteristics also add flexibility and extensibility to the system, since new types of parts may be introduced and old part specifications changed without altering the previously

defined data and process descriptions. And, a significant advantage which has already been identified in earlier paragraphs is the ability to directly map real objects to program objects giving the latter an intuitive touch.

In any software engineering methodology, the basic idea is to transform a model of the system from the problem space (real-world model) to a model of the system in the solution space (a programming model). This transition process is relatively smoother and more gradual in object-oriented systems than in systems developed through any other development process[2]. This is because the design paradigm used in these systems provides for a consistent representation of the system in terms of the real-world objects during every stage of software development.

3. Object-Oriented Design: The Basic Concepts

We start the design by identifying as entities the key absractions in the problem domain that will play a role in the application. The entities are identified based on the characteristics that they exhibit. The following is a collection of classification characteristics of entities, proposed by different authors: Tangible things, Events, Roles, Concepts, Organizations, External Systems, Locations, Devices [2]. Then, we specify the interaction among these entities, and associate with each entity the function performed by or on the entity. Following this, we analyze these functions to a level at which they are properly understood. By identifying key abstractions, it is possible to draw the system's boundary and filter out objects that do not belong within the system. The appropriate choice of objects depends, of course, on the purposes to which the application will be put and the granularity of information to be manipulated[2].

Abstractions are either discovered by recognizing them while examining the problem domain or created as useful artifacts of the design and implementation process [2]. For example, in the given CIM system, the entire system can be considered as an object at the highest level. Within it, each cell can be an object, and within each cell, each device can again be an object. In each cell, the cell controlling computer communicates with the devices through the ports on the data acquisition board specified. Ports can be objects too, because we can associate attributes such as port name, port address, direction of data transfer, etc. with them. These are abstractions of the

problem domain and will be identified as entities. The developer of such a system may invent additional objects such as coil, switch, pulse, etc., (different types of devices) which are artifacts of the particular design. It is difficult to state whether designers should begin abstracting from a lower or higher level. Most of the time, it becomes necessary that we should abstract from both levels and flip back and forth, and for this reason Booch [2] has termed the object–oriented design process as round–trip Gestalt design.

Very broadly, one can say that object–oriented design is an iterative process. At the end of each iteration, the design is examined for common attributes in a group of classes. The group of classes is then abstracted to a higher level to form a generalization. Similarly, specific needs of the application are identified, and new classes are derived from generic ones to define a specialization. This process is repeated until no new key abstractions can be identified, or when the identified classes and objects can be directly implemented in a straight forward manner or through available class libraries.

4. Description of the Environment and Behavior of CIM systems

The target environment is a product manufacturing system consisting of manufacturing units or cells networked together by computers and a data acquisition board for every cell that contains the interfacing electronics necessary to communicate between the manufacturing devices and the cell controlling computer. Each cell controller is networked to other cell controllers in order to provide remote programming and monitoring [7]. A typical manufacturing cell may consist of a Robot, a Conveyor, a CNC machine, Pallet Stops, Pallet Lifts, and a Photo sensor. The interfacing electronics serves to actuate and control these devices, and load programs written in the host command languages of these machines. The command files may be created from CAD/CAM software– one such software called SMARTCAM is available in the manufacturing engineering lab– and transferred directly to the cell controller. In addition, files may also be shared between programmers working from different remote stations. A schematic diagram of the cell control circuitry is given in Appendix A.

5. Language Design

Programming languages may be broadly classified as low-level, high-level, and fourth generation (4GL) languages. Low-level languages correspond directly to the hardware via binary machine code and assembly level opcodes, and often use registers to manipulate data. Programming a system using low-level languages increases its machine dependency. In high-level languages, most of the hardware details are transparent to the user, and the language is usually a set of symbols with which programmers are familiar, or in the worst case, can be learned relatively quicker than low-level languages and use the terminology of the problem domain. Usually, these are special-purpose languages.

Compilers are used to perform syntax checking and translate a program written in a programming language into machine specific executable code or an intermediate code, called i-code, which can be interpreted. The program may be executed only after the entire compilation is completed. Interpreters also perform syntax checking, but subsequently execute the code themselves immediately, i.e., after each line of the program is compiled, it is executed immediately without waiting for the remaining lines to be compiled. The CPL interpreter, however, does not perform any syntax checking, but can execute the intermediate code generated by the compiler in a step-wise fashion [9].

Programming languages are used to represent and describe algorithms, which are basically a sequence of steps. Until recently, the steps in programming languages were mathematical absractions of the actual operations in the real-world. Although, there are different categories of high-level languages available that serve software development in different domains-Fortran, Pascal, C, etc., for numerical computation; LISP, PROLOG, etc., for AI applications; COBOL for data processing applications; Ada for real-time systems programming, etc.- there is still a tight barrier that separates the programmer from the real system. With the advent of fourth-generation languages, special applications languages, and various redesigns of older languages, modern programming languages are better able to meet the specific needs of users in different domains. For example, languages such as SQL and QBE are specifically suited for data processing and database systems. Similarly, we have PostScript for document production, SIMULA and GPSS for simulation, etc.

The wide selection of languages currently available offer the user abstractions that provide a much better match for problems occurring in the real world.

Programming in low-level languages requires knowledge of the hardware that the system resides on and may even involve coding in machine language. System development of this nature is recommended only for embedded systems where the hardware and software machinery interface with real-world devices, and portability of the software is not an issue to be considered while building the system. The CIM system described in this document may be classified as an embedded system to a certain extent. Direct interfacing of the robot, lathe, conveyor, etc., through the data acquisition board introduces some amount of machine dependence that the programmers of this system cannot avoid. The task at hand is to develop and implement a language and an associated compiler that would mitigate if not eliminate the hardware dependency brought about by using device command languages of the robot and CNC machines for purposes of controlling the cell, and also be able to reflect the original manufacturing processes in the structure of a program written in the language. The language would be somewhere between a high-level language and a fourth generation langauge.

5.1 Design Issues

Some of the main issues concerning the design of a language are: the environment or domain in which it will be used, the terminology of the domain, the user group that it will serve, the functions that will be achieved by writing a program in the language, and restrictions on hardware platforms, if any, on which it will be available.

Given the goal that manufacturing engineering students should be relieved of requiring knowledge of command languages and hardware details, the simplest and most straight forward design decision in the language design process is that the language should be a high-level language. The aim is to provide as high a level of abstraction as possible so that the programmer can obtain a more accurate picture of the actual operations and behavior. This is possible if the constructs in the language corresponded directly to the abstractions in the real system. Instead of having to deal with data types and structures such as integers, floats, real, boolean, character strings, arrays, and

control constructs such as if-then, while-do, etc., it would be easier to program directly with abstractions such as cell, robot, ports, etc. The functions to be accomplished by the language would then be to declare values for the attributes or properties of each of these abstractions and define procedures on them to achieve a set of manufacturing processes. The language would then need to support predefined classes for each of the abstractions identified. However, supporting user-defined data types is beyond the scope of this project, but may be considered for future work.

5.2. CIM Operations and Objects

Operations in a cell are executed on devices by sending signals to them via ports on the data acquisition board. This scenario identifies three major categories of physical abstractions: ports, devices, and cells. Ports may be classified into serial ports, parallel ports, and ports that require an address specification. Similarly, the devices have been classified into coils, sensors, pulses, programmable, and wait based on the mechanism used to activate and deactivate them. The last type of device, wait, even though an operation, has been classified as a virtual device that implements a delay. Each device has a predefined set of operations that can be performed on it. A sequence of operations that define an identifiable process on a cell can be grouped together to give rise to another abstraction: Procedure. Each procedure can be executed on a cell, and entire procedures can be grouped together to achieve an integrated activity: a Program. The fundamental structures, therefore, that need to be explicitly defined by the programmer are: Ports, Devices, Cells, Procedures, and Programs. Defining the attributes and operations of these structures amounts to writing their class definitions, which is an easy task. The specific device types such as coil, sensor, etc., may be derived from the more basic Device class. In addition, specific events such as PalletLifted, PalletArrived, etc., as well as operations such as LatheStart, LatheStop, LatheHandshake, etc., may be regarded as virtual devices. A single physical device such as Lathe requires a strobe to start and stop it, but also requires a low or high signal as a hand shake before a program can be loaded into its memory. Since devices have been classified on the basis of the mechanism used to activate and deactivate them, it is imperative that CPL allows for defining some events

and operations also as devices. It is to be noted that the more basic data types such as integers, strings, etc. are implicit in the language, and are not treated as individual types.

In order to be able to define multiple ports, devices, and cells of a single type or class it is imperative that the language supports instantiation of classes. Although, CPL has been designed to allow object instantiation, changing the value of an attribute of an object is not permitted. However, currently, there is no apparent need for changing the characteristics of an object once its attributes have been assigned certain values. CPL is, therefore, a language without any variables in one sense.

Sequential operations may be implemented by grouping them together in procedures. Constructs for repetition are required to handle situations where a set of operations may need to be repeated in order to produce many parts. To implement this, procedures may be made to repeat *n* number of times by specifying an additional clause that gives the number of times that a procedure should repeat. If suppose, a storage cell executes a procedure that essentially loads parts onto the conveyor, and has to quit loading only when there is a signal from the manufacturing cell telling it to do so. In order to implement this, we need another clause to be added to the procedure construct that specifies a condition that should be fulfilled for the program to stop repeatedly executing a procedure. This clause is similar to the REPEAT....UNTIL statement in Pascal. At this point, a need to have statements that incorporate alternate constructs has not been identified, and therefore does not exist in the language definition.

Originally, devices were activated through individual commands in BASICA which required the specification of the port address, bit number, and the value for the bit in every command. This is a very cumbersome process, especially, if a PRINT statement is necessary for every command line that is to be sent to the programmable devices. Instead, a set of operations such as On, Off, WaitOn, WaitOff, Send, Do, etc., may be defined for devices, and the programmer may use them in conjunction with the device name to instruct the computer to turn On or Off a device. The advantage of this alternative is that the programmer is able to visualize the actual operations being performed with this kind of terminology, and also the command files that are created through the CAD/CAM software may be directly loaded onto

the programmable devices by using the Send and Do operations. CPL has been designed in such a way that it is possible to conceptualize an entire manufacturing process by following through the procedures, since each statement in a procedure corresponds to a single perceivable step in the actual process. This makes verifying a program a simple and easy task.

5.3. Classification of CPL

Programming languages roughly fall into one of the following categories: imperative languages, functional languages, and logic languages[8]. The fundamental operations of CPL are not dependent on the factors that characterize these types of languages. Another classification, categorizes languages as either procedural or declarative. The individual statements in CPL have a declarative tone, but it is also possible to explicitly specify an order of execution. Since all of the operations in CPL are performed on whole objects, it is best to classify CPL as an object–oriented language.

5.4. Formal Definition of CPL

A language definition serves to describe the syntax and semantics of a language. The syntax of a language defines the sequences of characters that make up a valid program in that language[8]. The semantics of the language describes the consequences of executing a valid program by describing the "meaning" of the syntax of the language. A formal definition of the language is essential to eliminate ambiguities in its syntax and semantics so that it serves as a precise definition for the compiler writer of the language. For this purpose, a popular meta–language, the Backus–Naur Form or Backus Normal Form (BNF) is used. A formal language specification is also called the grammar for the language. The grammar for its syntax is given in appendix D of this paper.

The class templates given in Appendix B serve to define the attributes and interface of CPL language and compiler classes, and also represent the various possible states of an object of the class. The inter–relationships between the predefined classes in CPL are described in appendix C in an object oriented notation that conforms to Booch, 1987 [2]. Appendix E contains the source code for the compiler implementation of CPL. The class templates are not actual class definitions; they are created in the initial stages of implementation in order

to serve as a starting point, and to give an idea of the attributes that characterize each class. Changes to the original class templates have been directly incorporated into the source code.

5.5. Grammar for CPL

CPL has been defined using a LL(1) grammar, where the next production rule to be applied is determined by examining the next input symbol. Source code symbols are examined left to right, and the leftmost derivation of a production rule is constructed in reverse. The definition of CPL in a LL(1) grammar makes the parser for its compiler implicitly top-down. Top-down parsers are simple and efficient tools that may be written by hand. Also, these parsers do not have to perform backtracking, since at every stage of the parsing process, the next production to be applied can be determined uniquely.

Implementing CPL requires developing a grammar that gives the definition of its syntax, and designing and implementing a compiler to translate CPL statements into executable or interpretable code. The following sections elaborate on the syntax of CPL and describe the design of a CPL compiler.

6. CPL Language Description

CPL does not hide all of the hardware details. In order to use CPL, the user is still required to know the particular hardware device and bits to which each device is interfaced. Also, the user must know the type of the device. Finally, individual cell components such as robots and CNC machines will have to be programmed in their host languages. One advantage, however, is that the programmer has full control of the operations, and can communicate with the individual devices even after the programs have been loaded into their memories.

It is difficult to differentiate between the data types and control constructs of CPL, since all constructs follow a declarative style of specification. Nevertheless, one distinguishing feature is that a control construct specifies an operation to be performed, whereas a data type declaration specifies the values of the attributes.

A CPL program consists of five major sections: port declarations, device declarations, cell declarations, procedure declarations, and program declarations. The following subsections elaborate on each of these.

6.1. Object Classes or Data Constructs in CPL

A CPL program has three major types

1. Ports: Used to name hardware interface ports.
2. Devices: Used to name individual cell devices, and assign ports and bit numbers
3. Cells: Used to declare network addresses of cell controlling computer.

In CPL, the instantiation of an object to be of a particular class determines the characteristics and the operations that can be performed on that object. All object classes are composite data types and a declaration of an object to be of a type or class also assigns values to the attributes of the type. So, a dynamic change in the value of an object's attributes is not possible.

6.1.1. Port Declarations

The port declaration section is used to instantiate port objects and to assign them physical port addresses on the cell controller PC. The declarations are made within a PORTS.... END block. Following the keyword PORTS is a series of individual port declarations. The syntax of a port declaration is as follows

< port_variable > (< port_address > < direction >) | (< port_name > < baudrate > < data_bits > < stop_bits > < parity >):

The *port_variable* can be any user defined identifier consisting of a maximum of 31 characters. The identifier can consist of alphabetic characters, digits and underscores up to a maximum of 31 characters. The *port_address* should be a physical port address and the *direction* is either INPUT or OUTPUT depending on whether the port is used to send or receive signals; the default direction is INPUT. If the port is a serial port, the *port_name* should be one of the serial ports COM1: or COM2: followed by *baudrate*, number of *data bits*, number of *stop bits*, and the type of *parity*. An example port declaration section follows.

Ports

PortA 64259 Output;

```
    PortB  64256  Input;
    PortC  64257;
    Com1port COM1: 3600 7 1 1;
End
```

6.1.2. Device Declarations

The device declaration section is used to declare a device object and associate a port and bit number with it. The device types are predefined and correspond to the devices in the cell. We have not made provisions for including user-defined device types in the language, because at this juncture we do not anticipate such a need. The declaration block is bounded by the keywords DEVICES and END. The syntax for the device declaration is as follows.

< device_variable > < device_type > (< port_variable > [< bit_number >]) | < programmable_port > ;

The *device_variable* is a user defined identifier and the *device_type* is a keyword in the language. The *port_variable* should have been defined earlier in the port declaration section, and the *bit_number* is a constant between 0 and 7 and corresponds to a bit on the data acquisition board. For a programmable device type, the port name LPT1 is specified if a parallel port is to be used, and the port identifier that has been assigned one of the serial ports COM1: or COM2: is specified if a serial port is to be used. An example device declaration section is given below.

```
Devices
 PalletLiftup    Pulse    PortC  4;
 Conveyor        Coil     PortC  5;
 Robot           Programmable LPT1;
 Lathe           Programmable Com1port;
End
```

6.1.3. Cell Declarations

The cell declaration section is used to assign network addresses to cell names in order to provide for communication between cells. Declaring cell names makes it convenient to assign a procedure to a cell, and facilitates modular programming at a small scale. The syntax

of a cell declaration is as follows:

< cell_variable > < network_address >:

The *cell_variable* is like any other user defined identifier, and cannot exceed a maximum of 31 characters. The *network_address* is a pre-defined host name or network address that has been assigned to the cell controlling computer by the system administrator. An example of a cell declaration section is given below.

```
Cells
        ManufacCell     cimlab6;
        StorageCell     192.34.54.3;
End
```

6.2. Control Constructs in CPL

A CPL program has two basic control constructs:

1. Procedures: Contains the sequence of cell control operations executed on devices
2. Program: Collection of procedures to be executed on cells

6.2.1. Procedure Declarations

The next section in the program is the procedure section which consists of statement constructs. Each statement represents one device operation and directly corresponds to an actual operation of the real device. The syntax of a procedure statement is as follows.

< device_variable > . (< device_function > [< open_parenthesis > parameter { . . . .}. < close_parenthesis >]) |

< delay_time >

The *device_variable* is an identifier previously declared in the device declaration section. The *device_function* is predefined, and is a keyword in the language. Table 1.0 lists device types and valid functions for each device type. Function parameters are enclosed within parentheses and are separated by commas. As with devices and ports, the keywords PROCEDURE and END mark the beginning and end of a procedure block. Each device function is checked for validity against the list of permitted functions for that device. An example of the procedure section is given below.

```
Procedure
        Conveyor.On;
        Robot.Send("NT");
        Lathe.Do(Cutpart);
        Delay.1000;
End
```

TABLE 1.0

TYPES	VALID FUNCTIONS
COIL	ON, OFF
SENSOR	WAITON, WAITOFF
PULSE	STROBE
PROGRAMMABLE	SEND, DO
DELAY	MILLISECONDS

Students can program device objects with names that directly correspond to their real-world counterparts. The predefined device functions are named after the actual device operations. For example, a statement such as Conveyor.On is an instruction to switch on the conveyor and a statement such as PhotoCell.WaitOn is an instruction to wait for the photocell to be switched on. This way it is possible to write a program and visualize an entire production operation without actually performing it.

The user is, however, required to program a programmable device type using its host language. Commands to the PROGRAMMABLE device type can be given directly by passing them as parameters to a function or they can be stored in a separate file, and the file name passed as the parameter. For example, Robot is declared to be a programmable device, and the Send operation accepts a parameter which is a string and sends a command to the robot. The Do function on the other hand accepts an identifier that is the name of a file consisting of robot commands which are read and directly output to the robot.

6.2.2. Program Declaration

The program section is the last section in a CPL program, and is simply a series of statements arranged in a predetermined order by the programmer. Each procedure statement states what procedure is going to be executed on what cell, and specifies repetition clauses, if any. The program statements appear within a PROGRAM....END block. The syntax of a program statement is as follows:

< cell_name > . < procedure_name > [(< condition > | < repetition_times >)];

The *procedure_name* is the name of a procedure that has been defined previously. Similarly the *cell_name* is also the name of a cell that has already been declared. Repetition clauses, if any, must be specified either as a *condition*, or as the number of *repetition_times* that a procedure is to be executed. These are enclosed within parentheses. An example program declaration is given below.

```
Program
        ManufacCell.ProduceBody;
        StorageCell.StoreParts (ManufacCell.SignalOn);
        ManufacCell.ProduceBody(50);  * Repeat 50 times
        StorageCell.StoreParts(50);
End
```

In CPL, every statement excepting block markers ends with a semicolon. The programmer may insert comments in the program by preceding the comment with an asterisk. A valid program statement and a comment may be typed on the same line, but the comment should follow the program statement. The reverse is, however, not true, because all characters in a line following a comment character are ignored by the CPL compiler. The syntax of this language is kept simple and compact in order to make it more appealing to users. At the end of the compilation, a cross-reference output listing the cross referencing between various devices and ports is printed. Error handling is performed by an error object which prints out error messages with corresponding line numbers and error codes. An example of a complete CPL program is given in appendix E.

7. CPL Compiler Design

The CPL compiler has been designed using object-oriented principles as another goal of this project was to gain experience in using object-oriented design and programming. The compiler is a two part system consisting of a translator and an interpreter. The translator parses the source code, analyzes it and generates intermediate p-code or pseudo-code (not the English like statements that are more commonly called pseudo code, but intermediate code consisting of opcodes and data in ASCII) that are easily interpreted by the interpreter. The parsing is recursive-descent top-down with single character backtracking.

7.1. CPL Compiler Objects and Classes

The project discussed in this report is concerned with designing and implementing the translator portion of the compiler, and hence will concentrate on its description. In the design of the translator, the entire translator is considered as an object at the highest level. Other objects at the top level of abstraction include those defining the symbol table which consists of information about objects in the CPL program, the cross-references object that produces a cross-reference listing for ports and devices at the end of the compilation, and a function table that defines device function keywords, number of parameters to be passed for device functions, and the types of the parameters. When the translator is invoked, a CPL program is instantiated to be of a translator type object. So, in essence, a program parses itself, and within a program, each construct and data type parses itself in turn, and generates its own code. The parser though object-oriented, performs top-down parsing with single character backtracking.

At the next level of abstraction, a generic statement class serves to define the common properties of all statements. A token class serves as a generic class to define the attributes and methods of all tokens, and a character class represents the properties and operations that can be performed on characters. The class templates for the translator classes are defined in appendix B, and the inter-relationships between classes at different levels of abstraction represented by class diagrams are given in appendix C.

7.1.1. The Translate Object

The compilation process is started when the translator is invoked with a parameter specifying the name of the program file along with the full path name. The program is instantiated as a translate object that begins the translation process. The translate object initializes the environment by creating the symbol table object, the token object, the cross-reference object, the i-code list object, the character object, and the errors object. It first strips the extension off the source file name so that default extensions can be appended to it when the output i-code and cross reference files are to be created.

The translation begins with the Ports section, and once the keyword "Ports" has been recognized, control is passed to a newly created Ports object. A new ports object is created for every port variable identified within the Ports block. Control is returned to the translate object after every port variable has parsed itself. The translate object executes a loop passing control back and forth between itself and a Port object until all port variable declarations have been parsed. Any errors discovered during the parsing process will be displayed on the screen by the errors object at the end of the compilation.

The translate object now parses the "Devices" keyword, after which control is again passed to a newly created Device object. As with the ports, a new device object is created for every device declaration. Similarly, a newly created cell, procedure, and program object is invoked for every cell declaration, procedure declaration, and program declaration statement respectively, and each object parses itself. Each port and device object successfully parsed is entered into the symbol table. If the entire program has been parsed successfully, the translate object prints the code generated by the procedure and program sections and invokes the cross-reference object to print a cross-reference listing between the various ports and devices.

7.1.2. The Symbol Table Object

The symbol table is used to maintain records of the variables being defined and the values of the attributes of the variable. There are two basic functions of the Symbol Table: Search and Insert. The Search function searches the symbol table for the object with the specified variable name. If the object is not found, it is inserted as a new variable if the insert flag is On. If the Insert flag if Off, and the object is not found, an error message is generated. This is mainly useful while parsing device and procedure statements. The code segment below is helpful in illustrating this.

```
Ports
    PortC 62526 Input:
    PortA 62527 Output:
    .....
End

Devices
    Robot Programmable LPT1::
    Conveyor Coil PortC 3:
    LatheStarted Pulse PortC 0:
    PalletLifted Sensor PortA 4:
    .....
End
```

Figure 2.0 Example Code Segment to Illustrate the need for a Symbol Table

In the device declaration section, LatheStarted is assigned the port variable PortC. It is essential that PortC be declared in the Ports block, lest the device be assigned an undefined port. When the port name is encountered while parsing the LatheStarted device declaration, the symbol table is searched for the port variable with the insert flag turned off. If it is not found, an error message is generated; if found, parsing continues to the next step. Similarly, in each procedure statement, the symbol table is checked for the specified device, and in each program statement, it is checked for the procedure and the cell variables.

7.1.3. The Token Object

The parsing operations have been encapsulated into a single object called the token object. The token object has been designed to scan the input file and identify tokens using a finite state machine algorithm. There are six basic states that the token object can exist in:

Initial, Identifier, Integer, Quote, Comment, and Done. The Initial state is the starting state of the token object, and also the state before a token is parsed. The token object interfaces with the character object to obtain the characters from the source code file. Depending on whether the character is a letter, integer, double quote, comment character, or a newline, the object switches into the Identifier, Integer, Quote, Comment, or Linefeed state respectively. The state transitions of the token object are shown in Figure 3.0 below. When a token has been identified, the token object is placed in the Done state. This indicates that it is ready for the next token to be parsed.

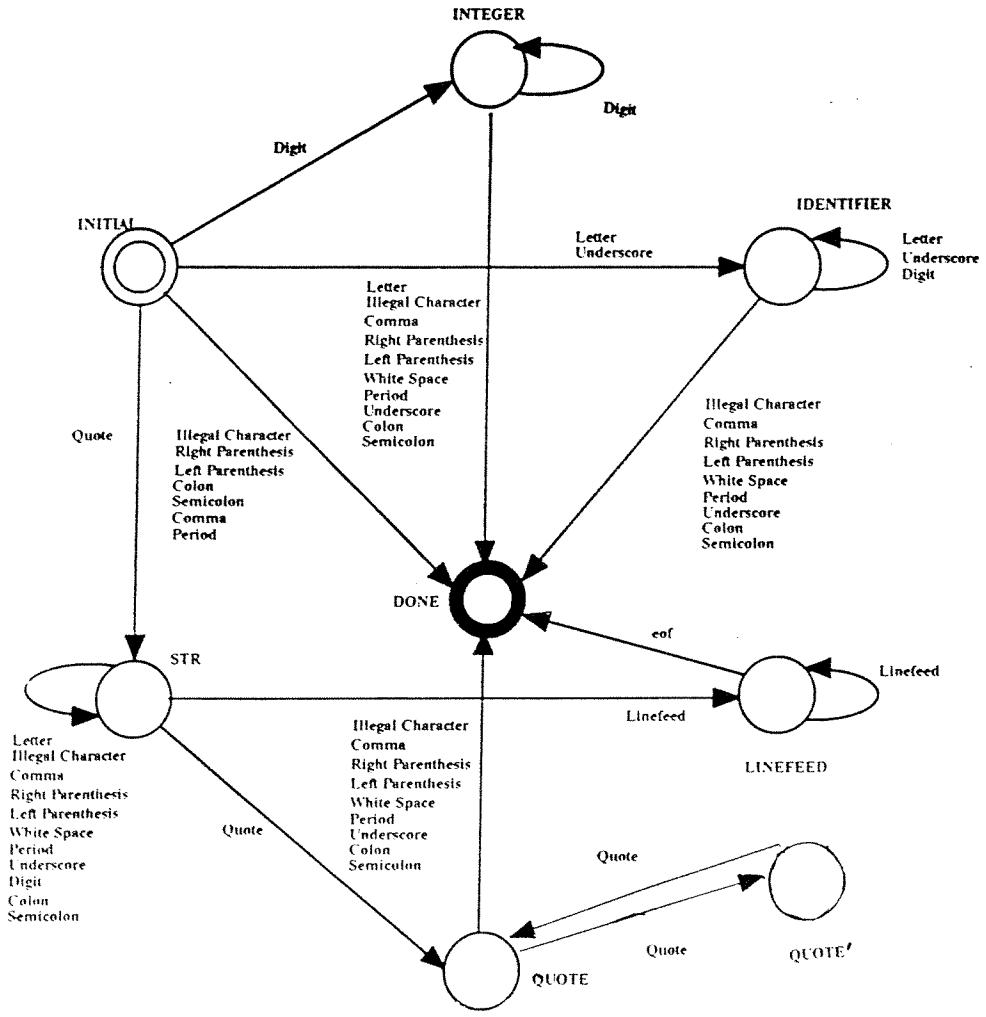


Figure 3.0 State Transition Diagram for the Token Object

7.1.4. The Character Object

A character that indicates a comment in the program has been introduced in CPL to facilitate source code documentation. In order to speed up parsing, it is necessary to ignore all characters on a line that follow an asterisk, the comment character. The functions of reading the source code lines from the program file and ignoring comments have been wrapped together into the character object. In addition, the character object implements single character backtracking with the help of a flag. If the flag is set, the previous character is returned to the client object; otherwise, the next character is returned.

7.1.5. The Function Table Object

This object houses information about the functions of each predefined device type in CPL. The function name, instruction opcode, number of parameters, and the type of each parameter are hard–coded into the function table object. This is not the best manner to make information about predefined functions available to the compiler. An alternative approach will have to be conceived in the future.

7.1.6. The Cross Reference Object

Although, the compilation of a CPL program is successful, there may be some errors that can only be discovered at run time. For example, an incorrect bit or port may be assigned to a device. This may lead to unpredictable results at run–time. A list of the objects cross referenced in the program is produced to aid the programmer in debugging, especially, of run–time errors. The cross reference object performs these functions and generates two lists, one for ports and device variables, and the other for device types and variables. The first list contains, for each port, the device name, the device type, and bit number of all devices using that port, and the second list contains the same information for each device type. Figure 4.0 is a cross–reference listing produced at the end of compiling the sample CPL program given in appendix E.

7.1.7. The Statement Class

The main syntax of CPL is derived from the statement class. This class serves as a template from which the different types of data types and control constructs of CPL are

Figure 4.0 Sample Cross-reference Listing

Cross references Between Ports and Devices

PortC	PalletLiftUp	Pulse	4
	Conveyor	Coil	5
	ChuckOpen	Pulse	1
	LatheG66inp	Pulse	0
	LatheStart	Pulse	2
	PalletStops	Coil	0
	ChuckClose	Pulse	0
	PalletLiftDown	Pulse	6
	LatheHandshk	Pulse	3
PortA	PhotoCell	Sensor	7
	PalletArrived	Sensor	6
	LatheStop	Sensor	4
	PalletLifted	Sensor	5
	LatheRunning	Sensor	2
Com1port	Lathe	Programmable	

Cross references Between Device Types and Devices

Pulse	PalletLiftUp	PortC	4
	ChuckOpen	PortC	1
	LatheG66inp	PortC	0
	LatheStart	PortC	2
	ChuckClose	PortC	0
	PalletLiftDown	PortC	6
	LatheHandshk	PortC	3
Coil	Conveyor	PortC	5
	PalletStops	PortC	0
Sensor	PhotoCell	PortA	7
	PalletArrived	PortA	6
	LatheStop	PortA	4
	PalletLifted	PortA	5
	LatheRunning	PortA	2
Programmable	Robot	LPT1:	
	Lathe	Com1port	

derived. All types of declarations in CPL are considered to be statements. The hierarchical breakdown of different types of statements is given in figure 5.0. The statement class per se is never instantiated during execution, but objects are created for the classes derived from it.

The Procedure and Program objects will have to generate code for each of the operations stated in their respective sections in addition to parsing each statement.

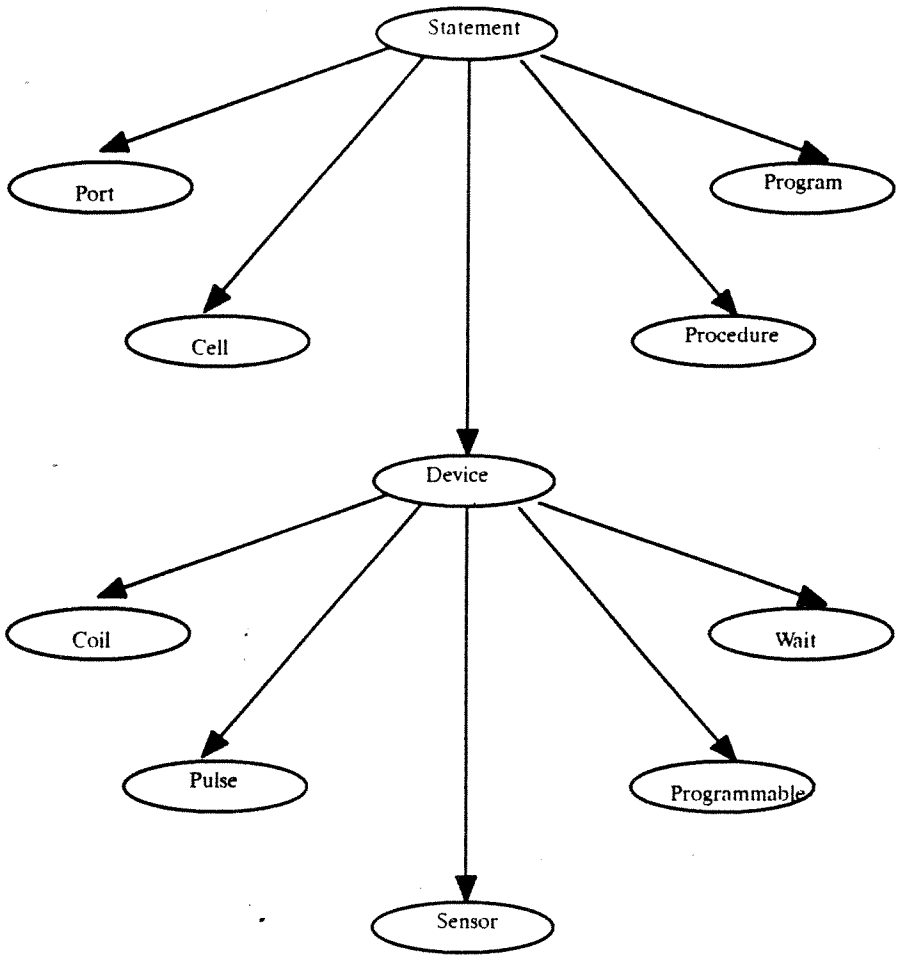


Figure 5.0 Hierarchical Breakdown of Statement Classes

7.1.8. The Errors Object

Whenever an error is discovered during compilation, the error object is invoked with an error code which is used to retrieve the corresponding error message from an error database. The idea behind maintaining a separate database for error messages was that new messages could be added and old ones deleted or changed without having to tamper with the compiler. Addition of new messages, however, requires that the new error code be included in the translator definitions file. Each error message is accompanied with a code that indicates the type of the error and the line number of the line in which the error was found. The code

helps the programmer in locating a description of the error message in the user manual that gives the possible causes for the error and suggests remedies to rectify it. There are three types of errors: Syntax error, Warning Error, and Fatal Error. The errors object keeps a count of the number of errors discovered and terminates compilation if more than a set maximum is found. However, the program is terminated immediately if a fatal error is discovered.

8. Benefits of CPL

The development of CPL has paved the way to an unified framework for enforcing efficient automation strategies in the CIM laboratory at Miami University. With CPL, the tedious process of learning individual command languages of devices is eliminated, and instead a more concise mechanism of specifying operations is available to students. The object-oriented nature of the language renders simplicity, and makes it less error-prone. Previously, locating errors in a command language program was very difficult because of the cryptic nature of the code and the absence of debugging aids. Now, the CPL compiler [9] provides programmers with useful debugging information in the form of cross reference listings at the end of the compilation. Also, the user manual accompanying the compiler documentation lists descriptions of error messages generated by the compiler. The accompanying interpreter [9] provides debugging options which allow the user to trace and step through the program while executing it. The syntax of CPL is simple and comprises problem domain terminology which makes it a closer and better representation of the real world, and as a result makes it more appealing to the user.

9. Future Development

The following is a list of future tasks for effectively implementing cell programming using CPL.

1. Currently, the translator has a set of pre-existing types and does not permit the declaration of user-defined types in the language. This feature would have to be included in the language.

2. The language has no control structures for alternation and this feature, if necessary,

will have to be incorporated into the language.

3. Although, declaration sections for cells and programs have been developed in the language, they have not been implemented in the compiler. In order to be able to provide communication between different cells in a CIM environment, these constructs will have to be incorporated into the system.

4. In the current version of the translator, each object parses itself, but does not generate its own code. This change in the implementation should be incorporated in order to conform with the design of the translator.

5. It is hoped that an integrated programming environment consisting of an editor and a full-fledged debugger will be provided with the compiler to enhance the capabilities of the programmer. At present, a simple debugging option is provided that allows the user to trace and step through the program.

6. Another important task for future CPL enhancements is to eliminate the need to learn the command languages of individual devices and memorize port addresses and bit numbers. This would reduce the hardware dependency of CPL to a great extent.

10. Summary and Conclusions

The application of object-oriented methods to integrate disparate manufacturing units has resulted in a flexible user-oriented automation environment. The development of a special purpose high-level language to enforce virtual homogeneity among different manufacturing devices provides a convenient control platform from where integrated manufacturing operations may be executed and monitored. Previously, hardware and device dependent details were indistinguishable from the needs of the users and could not be avoided. With the use of CPL, hardware and individual device intricacies can be separated from programming needs, thereby, providing a device independent mechanism to execute manufacturing operations.

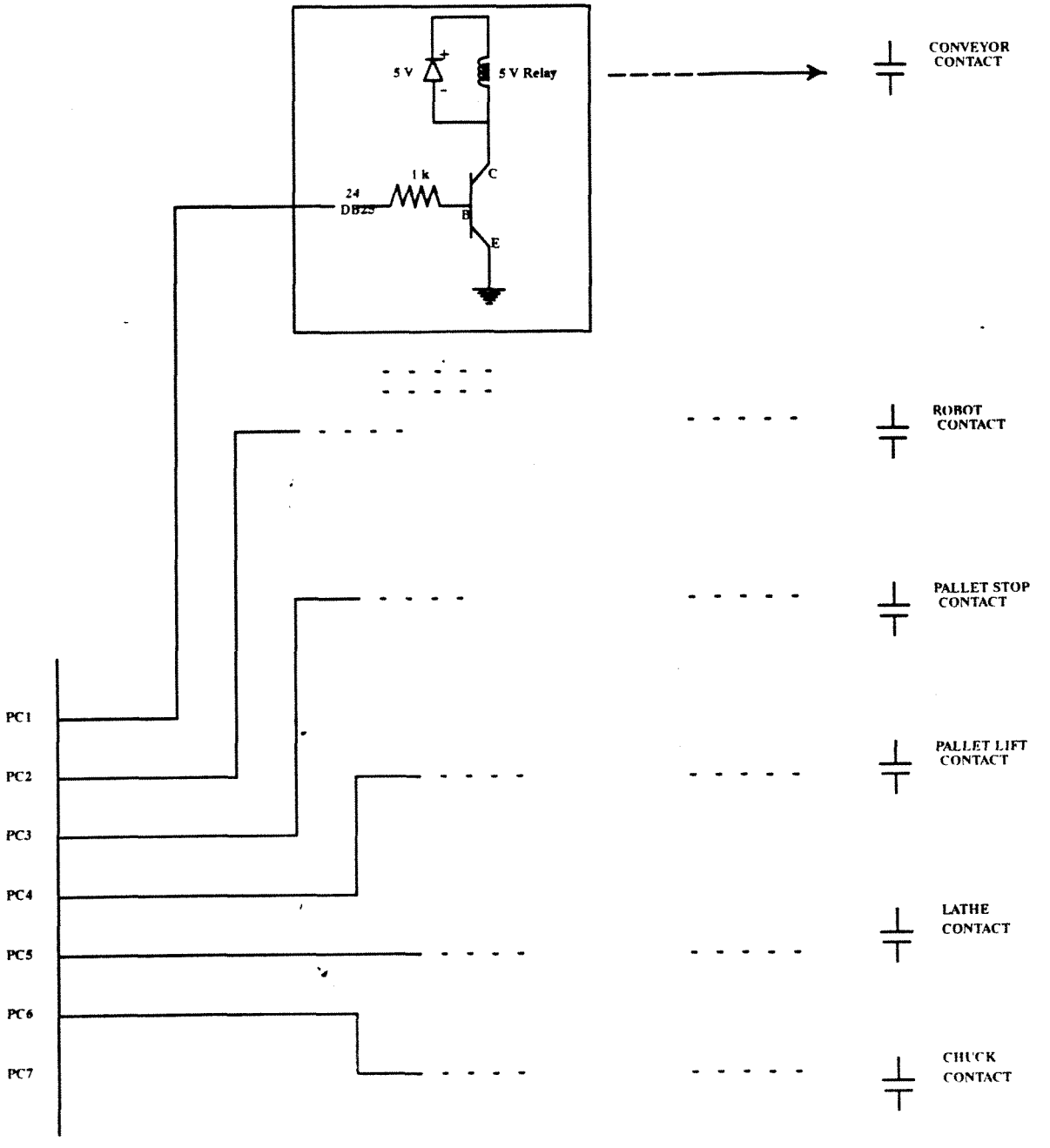
Developing a high-level language and an associated compiler was a useful exercise in studying language design criteria and compiler design techniques. Implementing the compiler in C++ resulted in good experience with programming in an object-oriented language. Over

all, it is now evident, that an interdisciplinary effort is necessary when applying computing and software technologies to other engineering fields.

References

1. Barry, B. M., SMALLTALK As a Development Environment For Integrated Manufacturing Systems, Proceedings of ICOOMS' 92, May 1992.
2. Booch, G., Object Oriented Design with Applications, *Benjamin/Cummings Publishing Company, Inc.*, 1991.
3. Cox, B. J., Object Oriented Programming: An Evolutionary Approach, Addison Wesley Publishing Company, April 1987.
4. Cox., B. J., Planning the Software Industrial Revolution, Proceedings of ICOOMS' 92, May 1992.
5. Hilfinger, P. N., Abstraction Mechanisms and Language Design, The MIT Press, 1983.
6. Nof. S. Y., Is all Manufacturing Object-Oriented?, Proceedings of ICOOMS' 92, May 1992..
7. Troy, A. D., Nugehally, M., Farooq., S., Hergert., D., Object-Oriented Flexible Manufacturing Systems at Miami University, Proceedings of ICOOMS' 92, May 1992.
8. Watson, D., High-Level Languages and Their Compilers, Addison Wesley Publishing Company, 1989.

APPENDIX A: Cell Control Circuit Schematic Diagram



APPENDIX B: Class Templates

During the first phase in Object-Oriented Design, entities in the system are identified as classes, and for each class attributes and functions are identified. These can then be presented in a semi-formal manner in class templates which have specific sections for attributes, interface, etc. Class templates are a useful documentation aid to the designer and help in the maintenance of the system.

This section provides the class templates for the classes identified in the design of the cell programming environment. Each template includes a brief informal description of the class, its status in the hierarchy, and the persistence of the instances of the class, i.e., whether the instances of the class are static (created once during an execution of the program), dynamic (created and destroyed more than once during a single run), or persistent (created and stored in secondary memory even after the program has ceased execution). In addition, attributes are listed in a subsection called Attributes, and the functions (the services offered by the class) are listed in a subsection called Interface. It should be noted that class templates are a semi-formal description of the class, and identifying attributes and functions does not amount to naming variables and functions. In the actual implementation, the variable and function names may be changed in order to comply with the syntax of the language chosen to implement the system. However, it is desirable to use the same names in order to maintain consistency throughout the development of the system.

In order to get an idea of the possible states of an object of a class, a state transition diagram is drawn. Circles represent states, and the arrows represent transitions. Labels on the arrows indicate the method or function that causes that state change, and labels on circles indicate the state of the object. The state labels are typed in uppercase and bold letters to differentiate them from the transition labels. The initial and final states are shown as double circles and a dashed circle indicates a temporary transition.

Port BASE STATIC

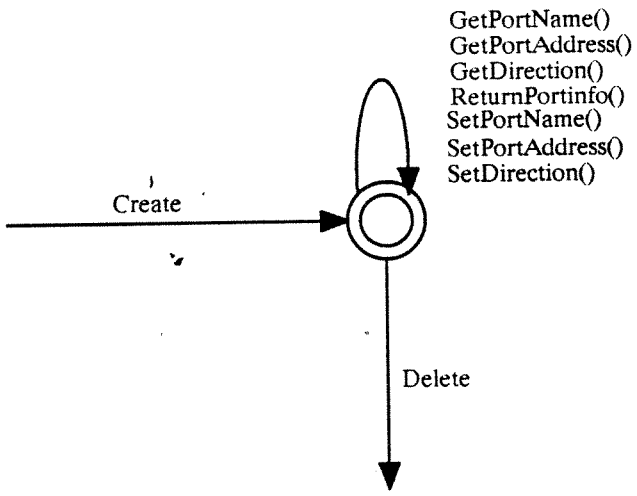
This class represents a single port declaration that contains information about a port variable which is one of the data types in CPL. In particular it contains a port identifier, the actual port address and the direction of communication (input or output)

Attributes

PortName
PortAddress
Direction

Interface

Create()
Delete()
GetPortName()
GetPortAddress()
GetDirection()
ReturnPortinfo()
SetPortName()
SetPortAddress()
SetDirection()



Device BASE STATIC

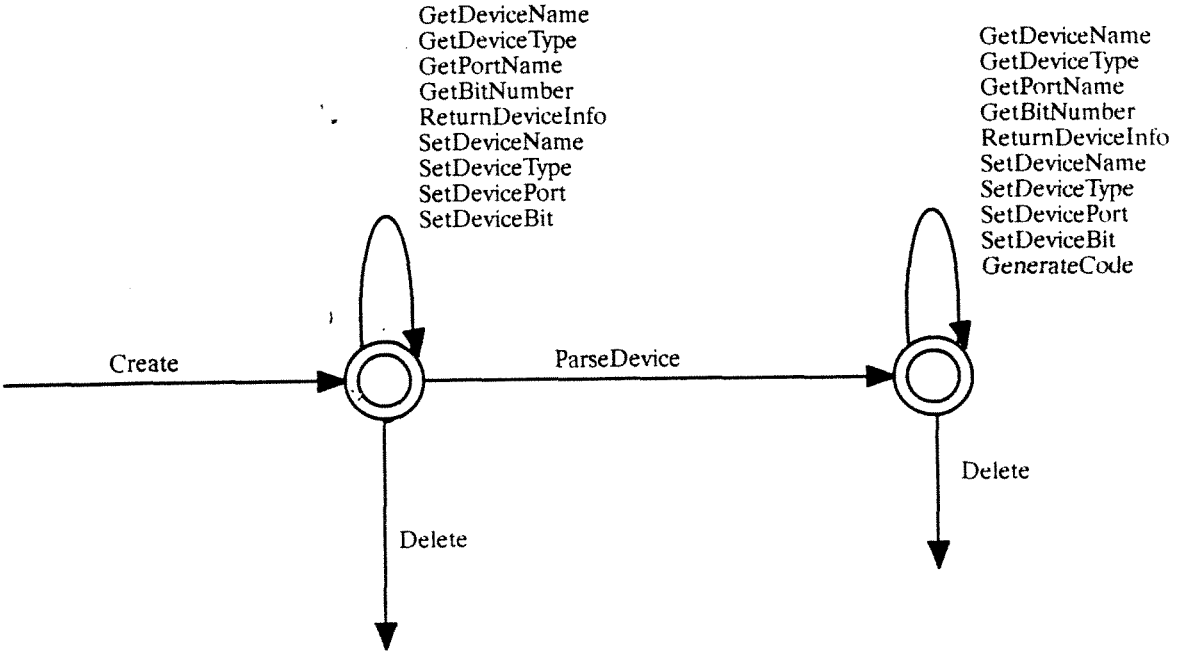
The device class represents a device declaration that contains information about a device variable which is one of the data types in CPL. In particular it contains the device identifier, the type of device, the port identifier, and the bit number on the port that is to be dedicated to the device.

Attributes

DeviceName
DeviceType
PortName
BitNumber

Interface

Create()
Delete()
GetDeviceName()
GetDeviceType()
GetPortName()
GetBitNumber()
ReturnDeviceInfo()
SetDeviceName()
SetDeviceType()
SetDevicePort()
SetDeviceBit()
ParseDevice()
GenerateCode()



Coil BASE STATIC

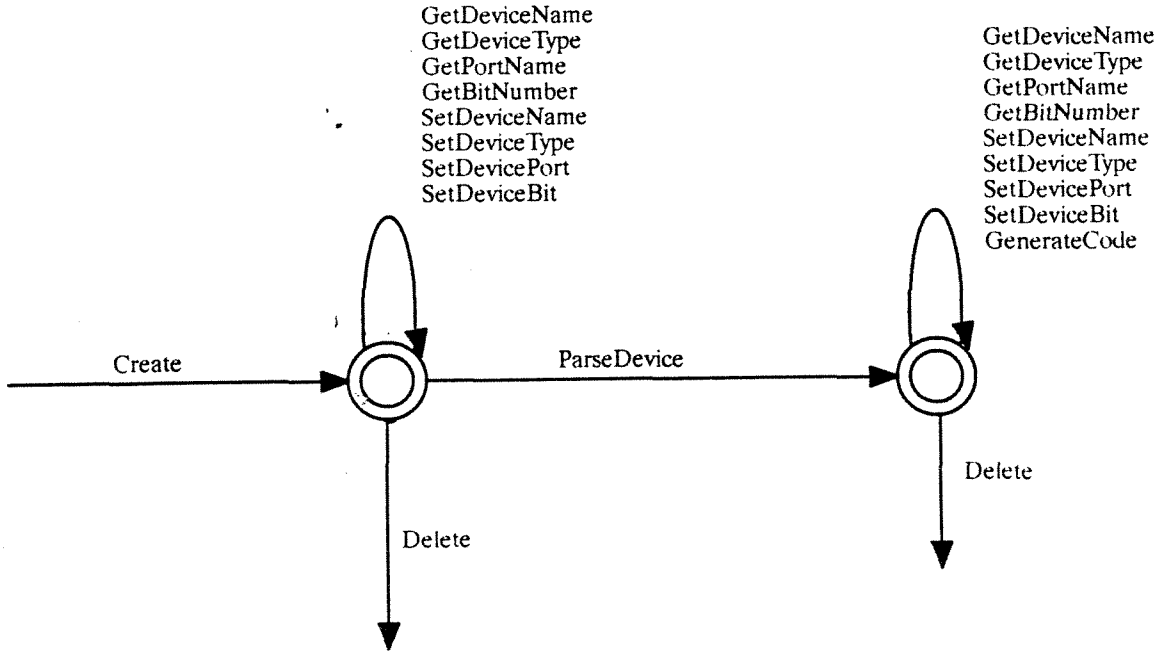
The coil type devices are actuated by solenoids, and hence the name. These devices require the device name, port name, and bit number. The conveyor and palletstops are examples of coil type device.

Attributes

DeviceName
DeviceType
PortName
BitNumber

Interface

Create()
Delete()
GetDeviceName()
GetDeviceType()
GetPortName()
GetBitNumber()
SetDeviceName()
SetDeviceType()
SetDevicePort()
SetDeviceBit()
ParseDevice()
GenerateCode()



Sensor BASE STATIC

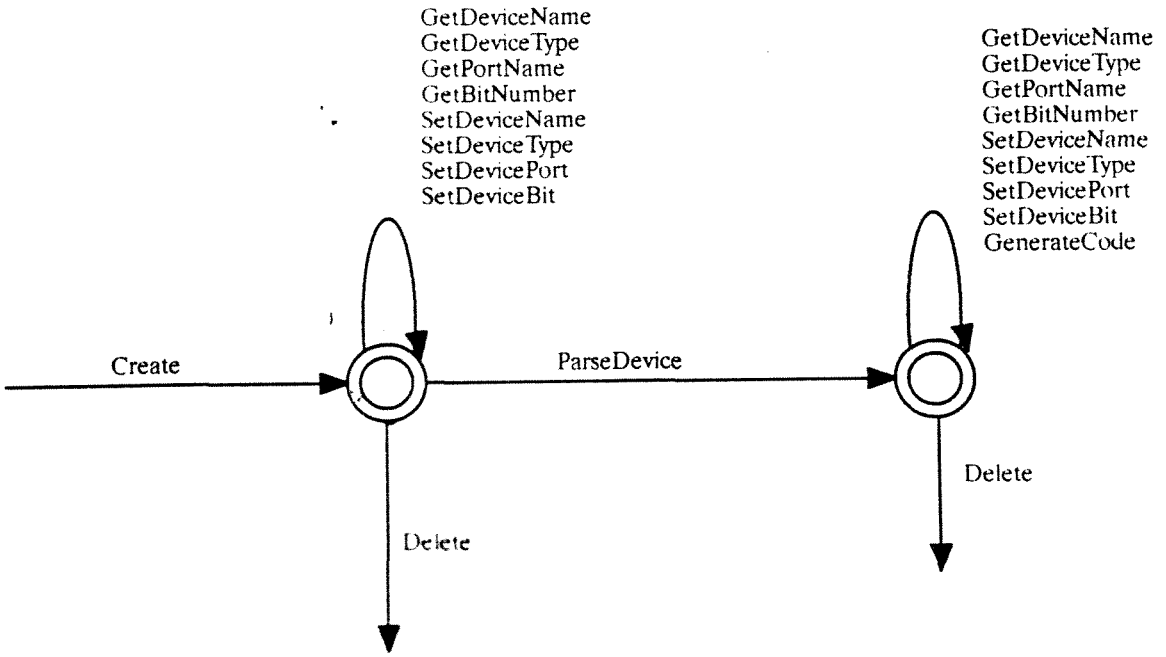
The sensor type of device is actuated using a 1 or 0. The photocell is an example of a sensor device. In addition, events such as palletlifted, palletarrived, etc can also be considered as virtual devices. Sensor devices require the device name, port name, and bit number for proper functioning..

Attributes

DeviceName
DeviceType
PortName
BitNumber

Interface

Create()
Delete()
GetDeviceName()
GetDeviceType()
GetPortName()
GetBitNumber()
SetDeviceName()
SetDeviceType()
SetDevicePort()
SetDeviceBit()
ParseDevice()
GenerateCode()



Pulse BASE STATIC

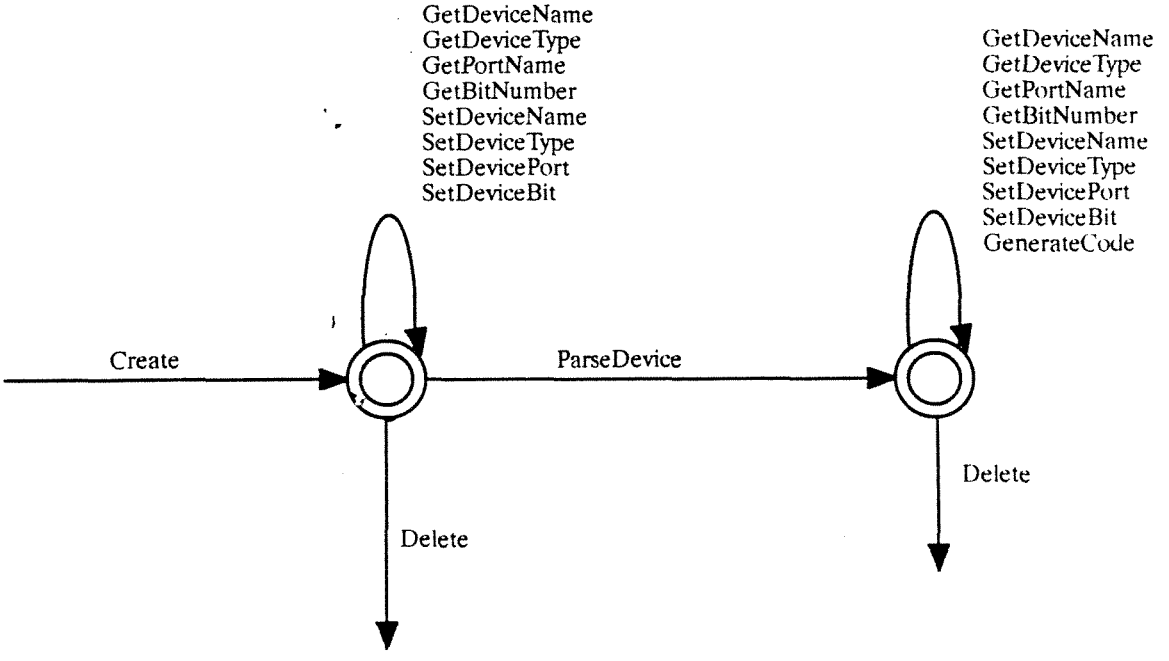
This class represents a pulse type device which is used for devices that require to be actuated and deactuated using a strobe. The Lathe is one such device. It is to be noted that the Lathe could also be a programmable device. The uniquely identifying feature between two different declarations for the same device is the port address and the bit number.

Attributes

DeviceName
DeviceType
PortName
BitNumber

Interface

Create()
Delete()
GetDeviceName()
GetDeviceType()
GetPortName()
GetBitNumber()
SetDeviceName()
SetDeviceType()
SetDevicePort()
SetDeviceBit()
ParseDevice()
GenerateCode()



Programmable BASE STATIC

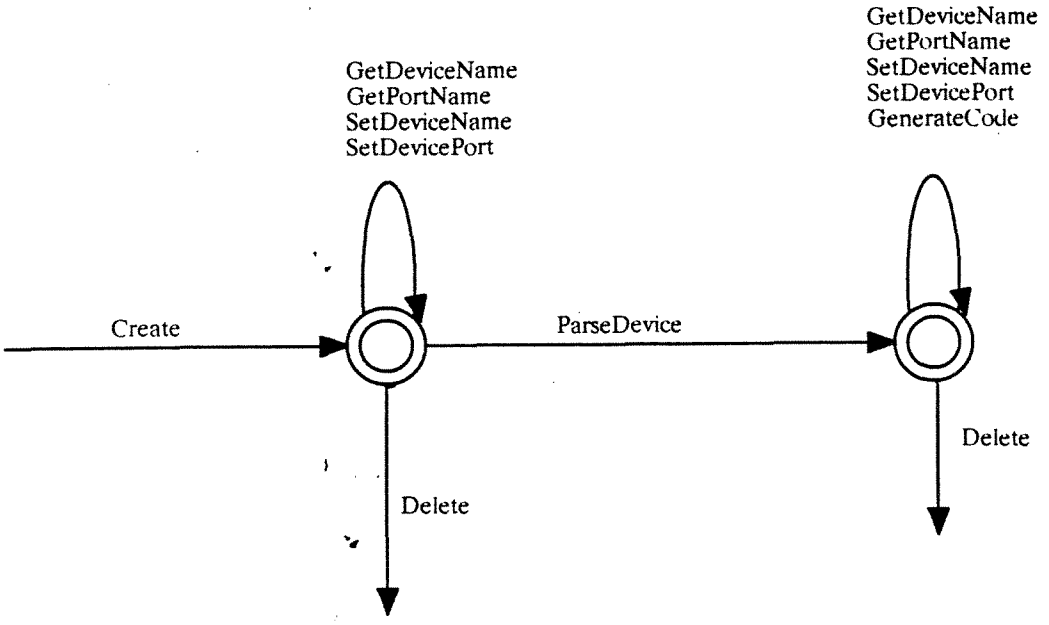
This clas represents a programmable type devicesuch as Lathe. Robot. etc. It consists of the device name and the name of the port used to communicate with this device

Attributes

DeviceName
PortName

Interface

Create()
Delete()
GetDeviceName()
GetPortName()
SetDeviceName()
SetDevicePort()
ParseDevice()
GenerateCode()



Wait BASE STATIC

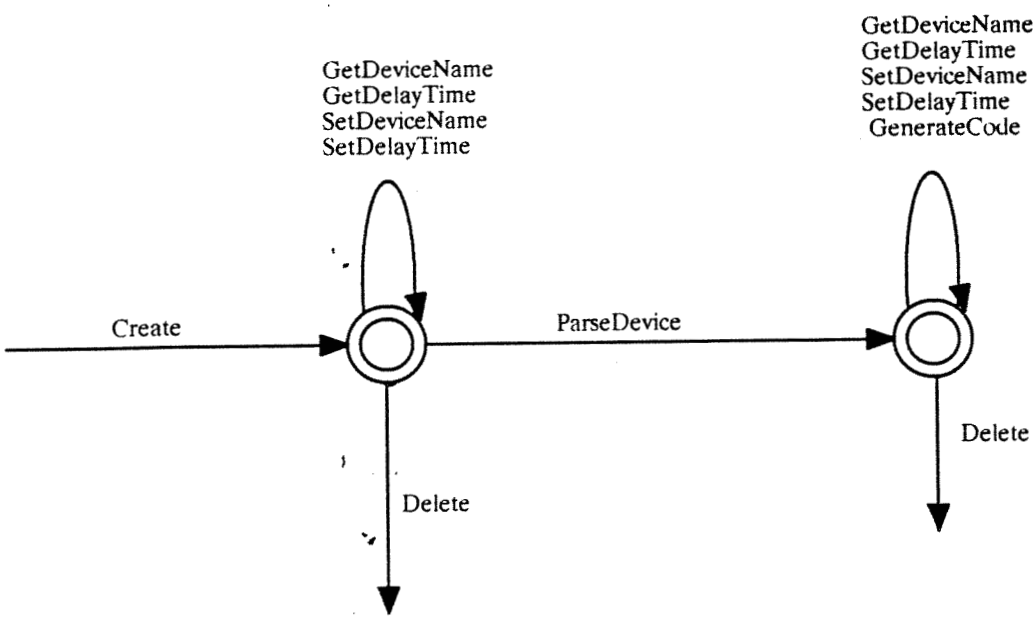
This class represents a wait type device and implements a delay loop. The delay time is given in milliseconds.

Attributes

DeviceName
DelayTime

Interface

Create()
Delete()
GetDeviceName()
GetDelayTime()
SetDeviceName()
SetDelayTime()
ParseDevice()
GenerateCode()



Procedure BASE STATIC

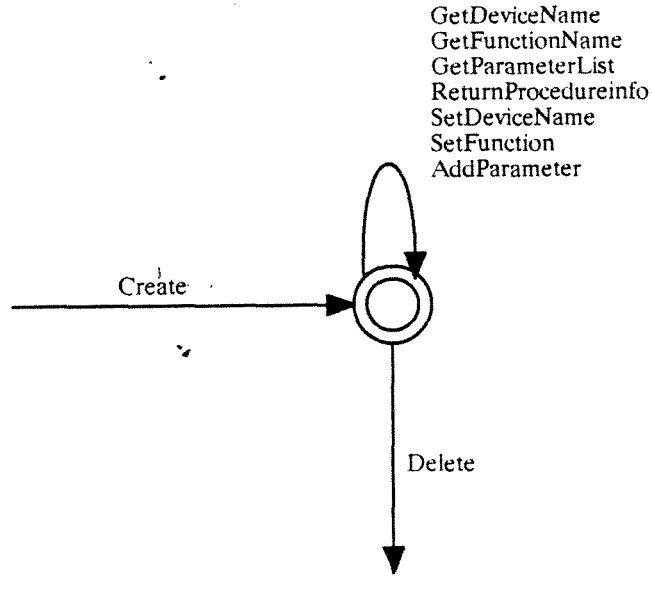
This class contains information about a procedure statement which is an expression in CPL. In particular it contains the device identifier, the operation to be performed on the device and a list of parameters for the operation.

Attributes

DeviceName
DeviceFunction
List of parameters
StatementNum

Interface

Create()
Delete()
GetDeviceName()
GetFunctionName()
GetParameterList()
ReturnProcedureinfo()
SetDeviceName()
SetFunction()
AddParameter()



Cell BASE STATIC

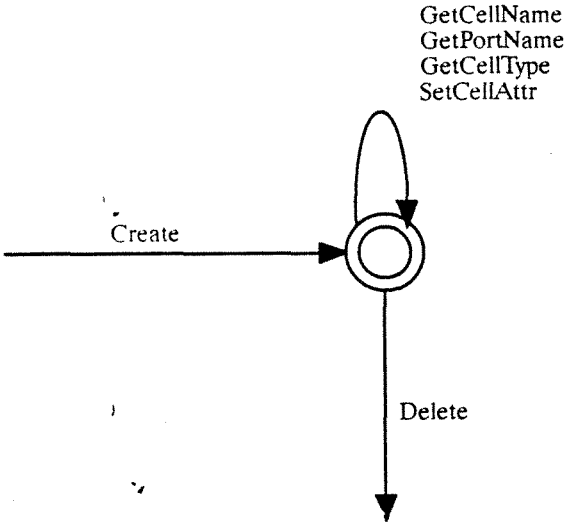
This class represents a cell declaration and contains information about a cell variable which is one of the data types in CPL. In particular it contains the cell identifier, the cell type, and the port address of the computer used to control the cell.

Attributes

CellName
CellType
PortName

Interface

Create()
Delete()
GetCellName()
GetPortName()
GetCellType()
SetCellAttr()



Program BASE STATIC

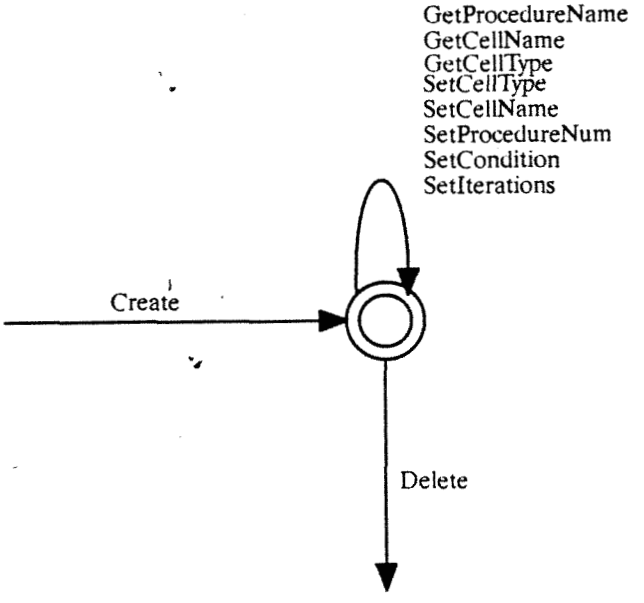
Represents a program statement which is an expression in CPL. In particular it contains the procedure name, the procedure number and the name of the cell to which the procedure is to be applied.

Attributes

ProcedureName
CellName
ProcedureNumber
ExecutionCondition
IterationNumber

Interface

Create()
Delete()
GetProcedureName()
GetCellName()
GetCellType()
SetCellType()
SetCellName()
SetProcedureNum()
SetCondition()
SetIterations()



TRANSLATOR BASE STATIC

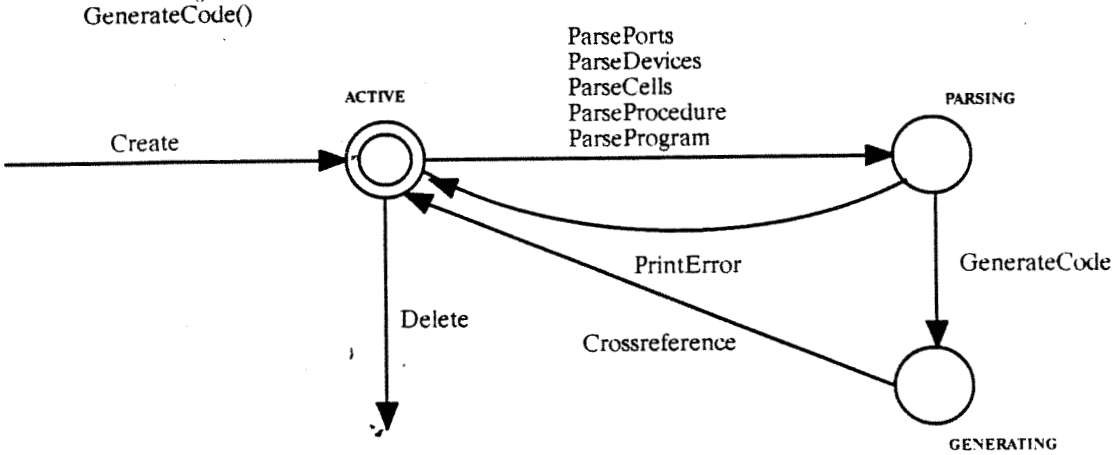
The main class of the translating system. Initializes the variables, open files, and initiates the parsing process.

Attributes

ProgramFile
Token
Character
Source Directory
Output Directory

Interface

Create()
Delete()
ParsePorts()
ParseDevices()
ParseCells()
ParseProcedure()
ParseProgram()
PrintError()
GenerateCode()



SYMBOLTABLE BASE STATIC

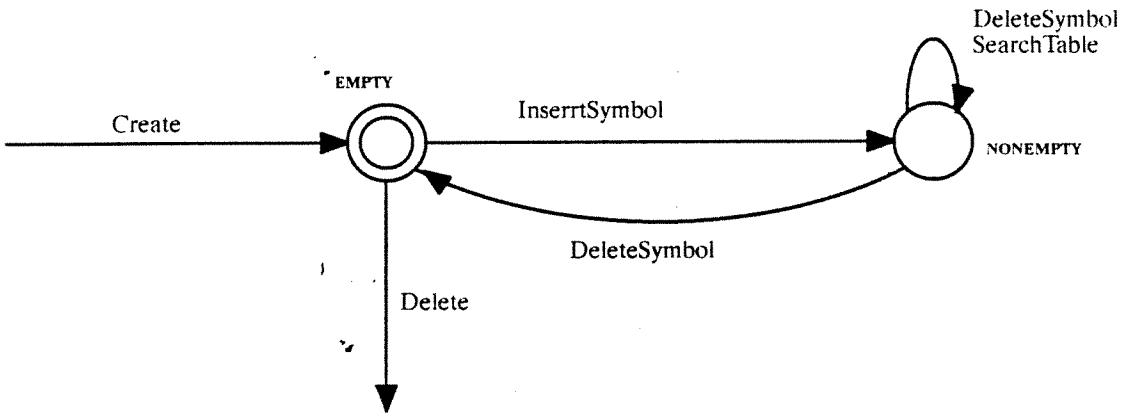
A table for storing information about port. device. and cell variables. The two mjor operations of the table are insert and search. Insert allows you to add a symbol into the table . and search allows you to search the table for the given symbol information.

Attributes

Symbol
Symbol type
Pointer to next symbol

Interface

Create()
Delete()
InsertSymbol)
ReturnSymbol)
ConvertSymbolPointer()
SearchTable
DeleteSymbol()



FUNCTIONTABLE BASE STATIC

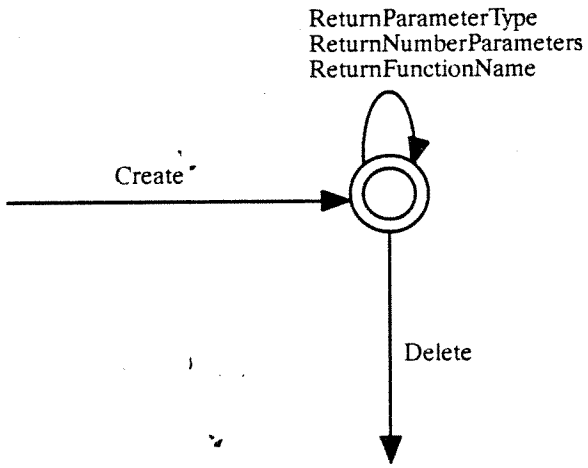
The function table is used to store information about CPL predefined functions. The number of parameters. the opcode. and the types of the parametrs are stored for each function.

Attributes

FunctionName
Number of Parameters
Parameter List
Opcode

Interface

Create()
Delete()
ReturnParameterType()
ReturnNumberParameters()
ReturnFunctionName()



CROSSREFERENCE BASE STATIC

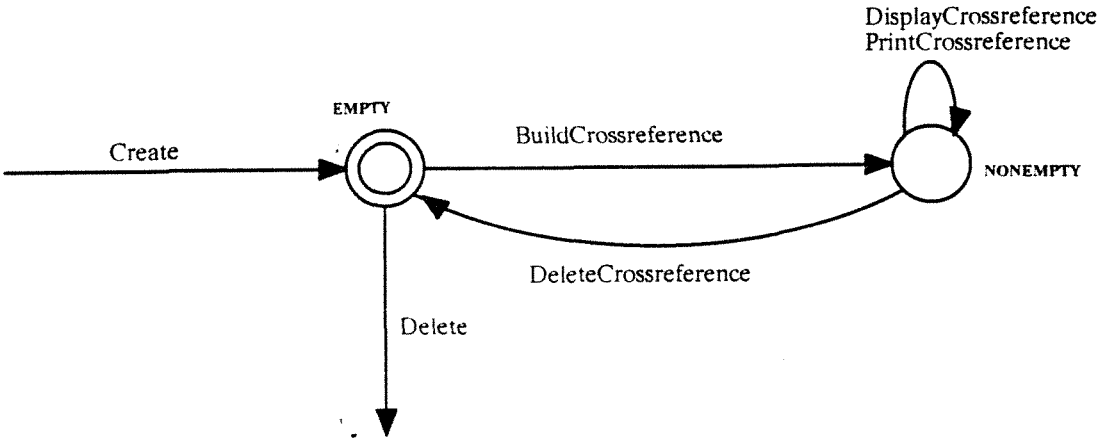
A class to put together all the crossreference information. This includes a list of the ports and the devices using each of the ports including the type of the device and the bit assigned for each port variable. A crossreference is created for the type of the device and thedevice variables declared to be of that type.

Attributes

PortCrossreference
DeviceCrossreference

Interface

Create()
Delete()
BuildCrossreference()
DisplayCrossreference()
PrintCrossreference()
DeleteCrossreference()



STATEMENT BASE STATIC

This generic class represents a statement as an object. This class will not be instantiated and only serves as a base class for specific statement types such as port statement, decive statement, etc., and hence there will not be any state transition diagram for it.

Attributes

StatementType

Interface

Create()
Delete()

TOKEN BASE STATIC

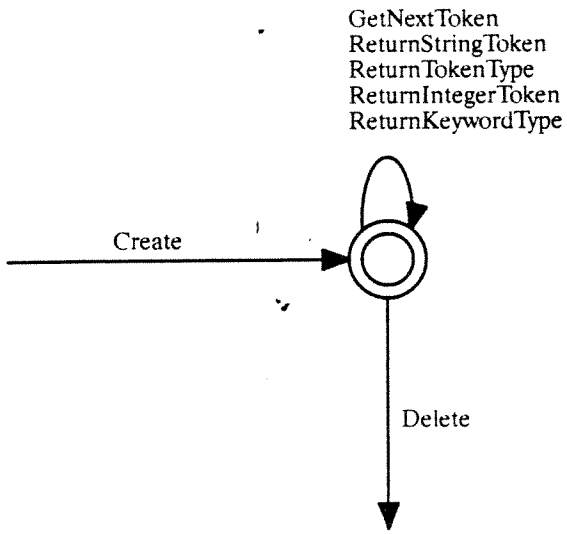
A token is a set of characters strung together. This class classifies tokens as integers, strings, keywords, special characters, illegal tokens, etc.

Attributes

Token Type
String Token
Integer Token
Keyword type

Interface

Create()
Delete()
GetNextToken()
ReturnStringToken)
ReturnTokenType()
ReturnIntegerToken()
ReturnKeywordType()



CHARACTER BASE STATIC

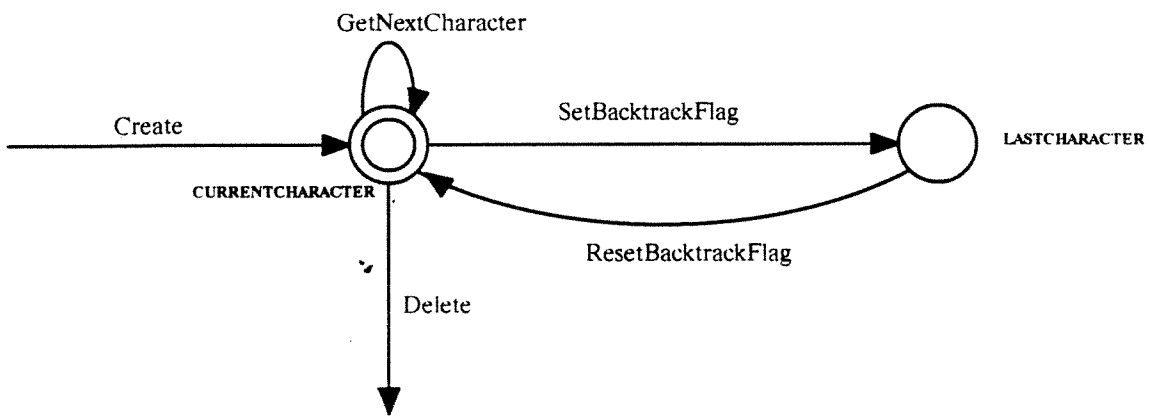
This class represents a character and all the attributes and operations associated with a character. It serves to scan the source code file for the next character and implements single character backtracking.

Attributes

CharacterScanned
BacktrackFlag

Interface

Create()
Delete()
GetNextCharacter()
SetBacktrackFlag()
ResetBacktrackFlag()



ERROR BASE STATIC

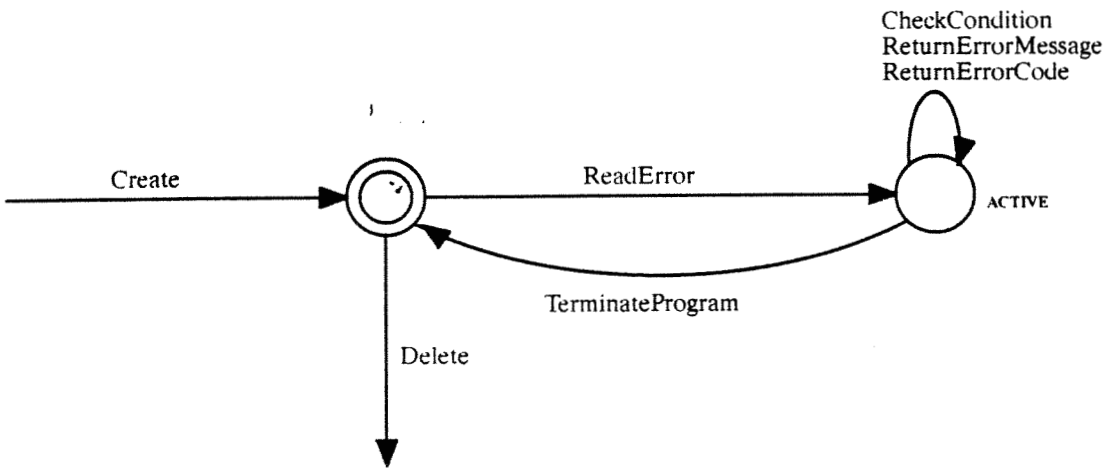
Represents the error database and is responsible for handling compile time errors. Contains an error database consisting of error codes and their corresponding messages. The error object reads errors from the database based on the error code passed to it. It checks for the error condition and either terminates the program or increments the error count depending upon the number of errors discovered or when a fatal error is encountered.

Attributes

ErrorType
ErrorFile
Number of Errors
Number of FatalErrors
Number of Warning Errors
ConditionType
ErrorMessage
ErrorCode

Interface

Create()
Delete()
ReadError()
CheckCondition)
ReturnErrorMessage()
ReturnErrorCode()
TerminateProgram()



APPENDIX C: Class Diagrams

Class diagrams depict the relationships between different classes. The notation used in class diagrams consists of dashed irregular shapes connected by arrows. The type of the arrow between any two classes represents the type of the relationship between them. One can represent four basic kinds of relationships: Inheritance, Instantiation, Containing, and Using

The notation consists of the following symbols:

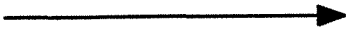
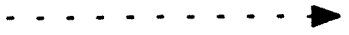
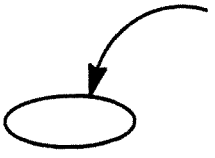
name	:	Label used to denote the name of the class
	:	Icon to denote a Class
	:	Inheritance Relationships– the class at the tail of the arrow is derived from or inherits the attributes and functions of the class at the head the arrow.
	:	Instantiation Relationship– the object at the tail of the arrow is an instance of the class at the head of the arrow
	:	Using Relationship– the class at the circle uses the services or functions of the class at the tail of the arrow in its implementation. The icon expands to a double line as it is made longer.
	:	Using Relationship– the class at the circle uses the services or functions of the class at the tail of the arrow in its interface.
	:	Containing Relationships– The class at the tail of the arrow contains the classes enclosed in the oval. The oval will have to be increased in size in order to accommodate the component classes.
	:	Icon to denote a class library
	:	Icon to denote a class inside a subsystem

Figure C.1 Using Relationships for Translator System

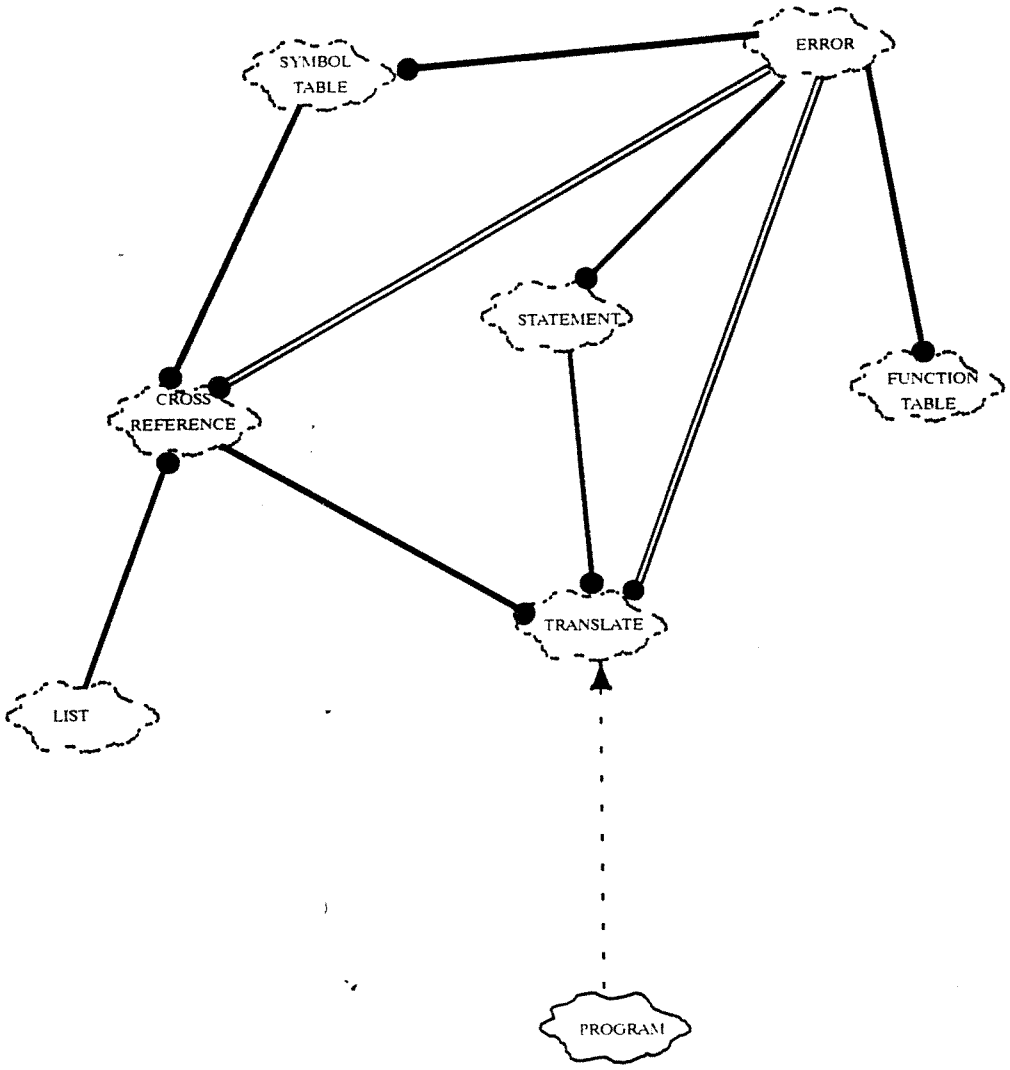


Figure C.2 Using Relationships for Translator System

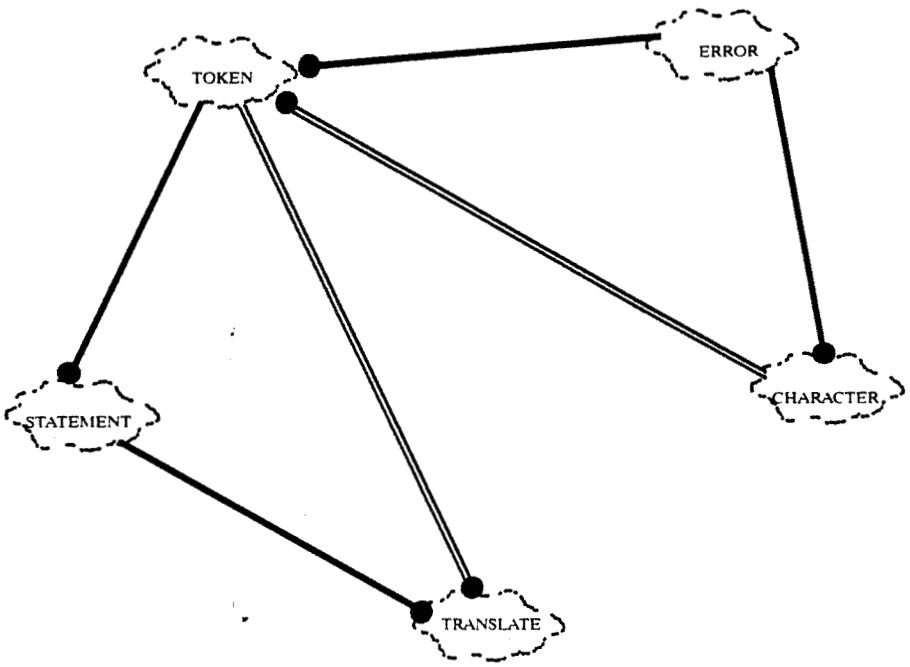


Figure C.3 Inheritance Relationships for Class Statement

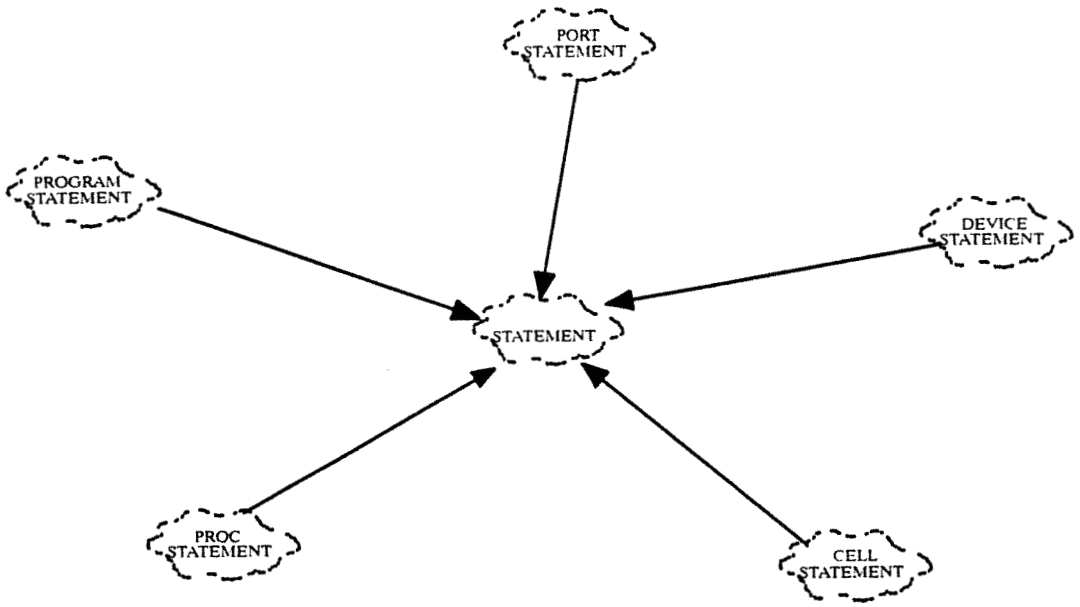


Figure C.4 Instantiation Relationships for Class Statement

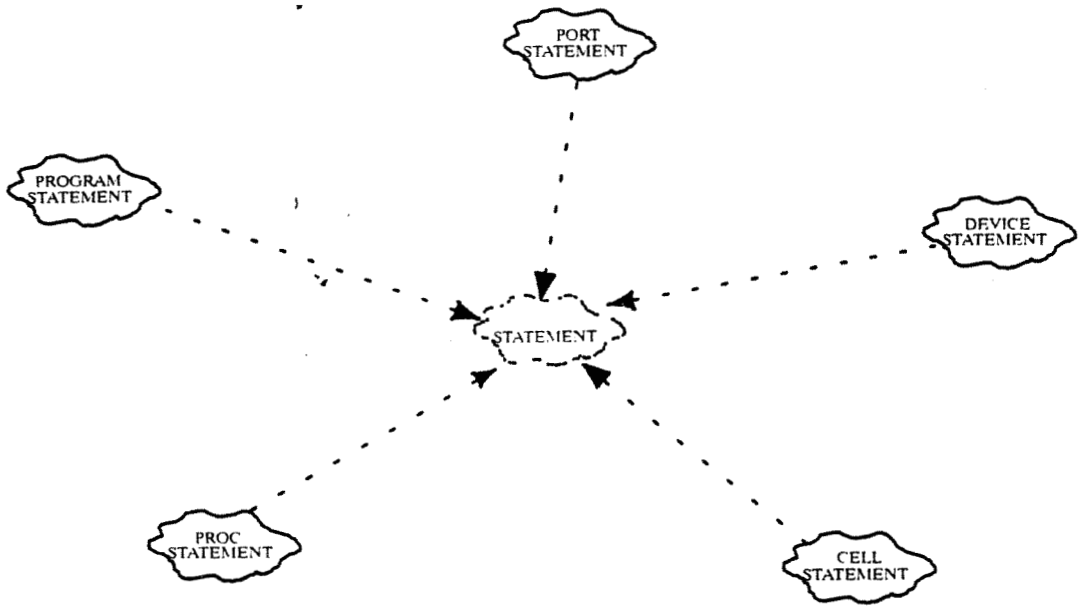


Figure C.5 Inheritance Relationships of Class Device Statement

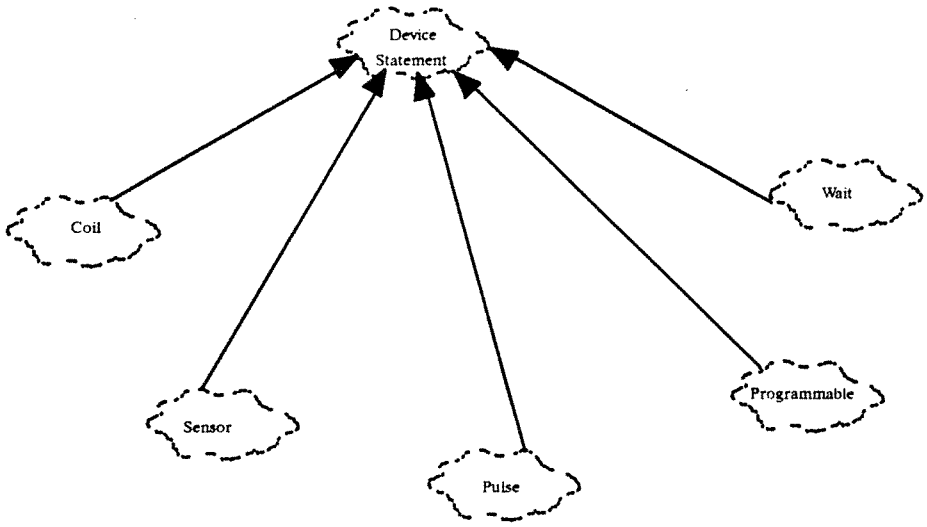
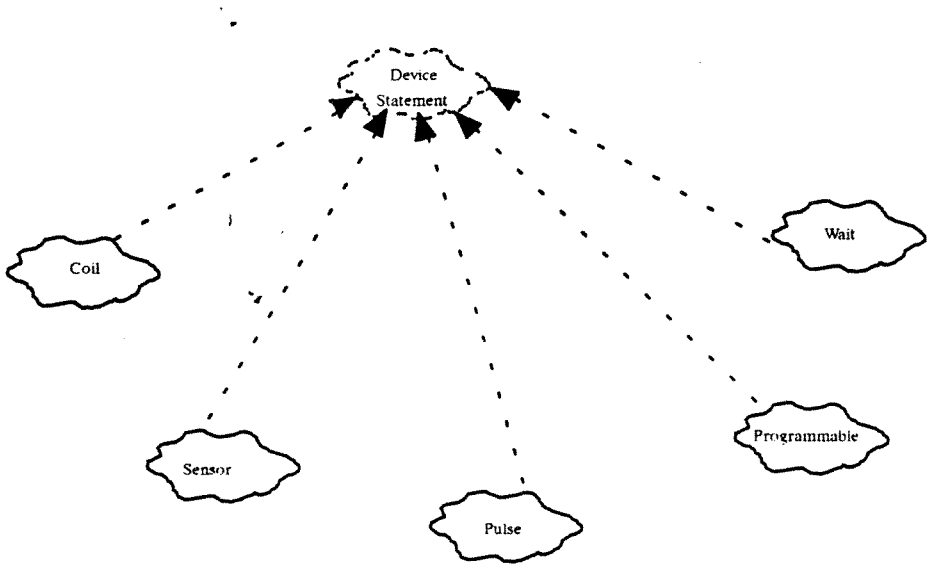


Figure C.6 Instantiation Relationships of Class Device Statement



APPENDIX D: CPL Grammar

The Cell Programming Language is designed as a context-free grammar and uses the Backus Naur notation for expressing the syntax. The grammar is free of ambiguity at present but shall be corrected for any ambiguities detected henceforth.

CPL is described as a set of production rules, each production rule consisting of a left hand side and a right-hand side, separated by an assignment operator. An example of a production rule is given below.

Production Rule : An example

<expr> -> <expr> + term> | <expr> - <term> | <term>

<term> -> 0|1|2|3|4|5|6|7|8|9

The left-hand side is always a non-terminal symbol. The right hand side may be a combination of non-terminals and terminals, only non-terminals, or only terminal symbols. The non-terminals are enclosed within the '<' and '>' symbols and the terminals are represented by constant values. The '—>' string serves as the assignment operator. Optional symbols may be enclosed within square brackets or braces '[' and ']'. Parenthesis are used to specify grouping of symbols, and when more than one symbol, separated by commas, is enclosed within '{' and '}', it means that at least one of the symbols or a group of symbols must be present. For example, in line 16 of the CPL grammar, the device_data must either consist of the port_name followed by a valid_bit or just a predefined_port. The parenthesis around port_name and valid_data indicates that these two are grouped and hence they go together. In this grammar, the language keywords, alphabets, special ASCII characters and digits are the terminal symbols. Currently, the language provides for sequence and repetition constructs, but also leaves room for adding alternative constructs. The CPL production rules and selection sets are shown in the following pages. Selection sets are sets of tokens that help determine the number of tokens necessary to apply the next production in a given rule. In this case, the a single token is sufficient to determine the next production to be applied. This makes the CPL grammar LL(1).

CPL Production Rules and Selection Sets

1.	<cpl_program>	:=	[<port_declarations>] <cell_declarations> <device_declarations> <procedure_declarations> <program_declarations>	[PORTS, DEVICES, CELLS, PROCEDURE, PROGRAM]
2.	<port_declarations>	:=	PORTS <port_stmtlist> END	[PORTS, LAMBDA]
3.	<cell_declarations>	:=	CELLS <cell_stmtlist> END	[CELLS]
4.	<device_declarations>	:=	DEVICES <device_stmtlist> END	[DEVICES]
5.	<procedure_declarations>	:=	PROCEDURE <procedure_name> <procedure_stmtlist> END	[PROCEDURE]
6.	<program_declarations>	:=	PROGRAM <program_stmtlist> END	[PROGRAM]
7.	<port_stmtlist>	:=	<port_stmt> [<port_stmtlist>]	[TIDENTIFIER, END]
8.	<port_stmt>	:=	<port_name> <port_address> <direction>	[TIDENTIFIER, TINTEGER, TKEYWORD]
		:=		
9.	<port_name>	:=	<identifier>	[TIDENTIFIER]
10.	<port_address>	:=	<integer>	[TINTEGER]
11.	<direction>	:=	[KINPUT] [KOUTPUT	
12.	<cell_stmtlist>	:=	<cell_stmt> [cell_stmtlist>]	[TIDENTIFIER, END]
13.	<cell_stmt>	:=	<cell_name> <cell_address>	[TIDENTIFIER, THOSTNAME, TNETADDRESS]
14.	<cell_name>	:=	<identifier>	[TIDENTIFIER]
15.	<cell_address>	:=	THOSTNAME [TNETADDRESS	
16.	<device_stmtlist>	:=	<device_stmt> [<device_stmtlist>]	
17.	<device_stmt>	:=	<device_name> <device_type> <device_data> TSEMICOLON	[TIDENTIFIER, TKEYWORD, TIDENTIFIER]
18.	<device_name>	:=	<identifier>	[TIDENTIFIER]
19.	<device_type>	:=	PROGRAMMABLE [<nonpgble_type>	[TKEYWORD, TIDENTIFIER]
20.	<device_data>	:=	[(<port_name> <valid_bit>), <predefined_port>]	[TIDENTIFER, TKEYWORD]
21.	<procedure_stmtlist>	:=	<procedure_stmt> [<procedure_stmtlist>]	[TIDENTIFIER, END]
22.	<procedure_stmt>	:=	<device_name> [<period> <device_func>] TSEMICOLON	[TIDENTIFIER, TKEYWORD, TDOT]
23.	<device_func>	:=	<pulse_func> [<coil_func> [<sensor_func> [<programmable_func> [<wait_time>	[KON [KOFF, KWAITON [KWAITOFF, KSTROBE, KDO [KSEND, TINTEGER]
24.	<procedure_name>	:=	<identifier>	[TIDENTIFIER]
24.	<program_stmtlist>	:=	<program_stmt> [<program_stmtlist>	[TIDENTIFIER, TKEYWORD]
25.	<program_stmt>	:=	<procedure_name> <cell_name> [(<repetition_clause>)]	[TIDENTIFIER, TLPAREN]
26.	<repetition_clause>	:=	<iterations> [<condition>	[TINTEGER, TIDENTIFIER]
27.	<iterations>	:=	<integer>	[TINTEGER]
28.	<condition>	:=	<cell_name> <period> <signal>	[TIDENTIFIER, TKEYWORD, TDOT]
29.	<signal>	:=	ON [OFF	
31.	<pulse_func>	:=	<period> KSTROBE	[TDOT, KSTROBE]
32.	<coil_func>	:=	<period> KON [KOFF	[TDOT, KON [KOFF]
33.	<sensor_func>	:=	<period> KWAITON [KWAITOFF	[TDOT, KWAITON [KWAITOFF]
34.	<programmable_func>	:=	<period> KSEND [KDO ({ <parameter>, . . . })	[TDOT, KSEND [KDO, TLPAREN]
35.	<parameter>	:=	<string> [<identifier>	[TSTRING, TIDENTIFIER]
36.	<wait_time>	:=	<period> <integer>	[TINTEGER]
37.	<nonpgble_type>	=	KCOIL [KSENSOR [KPULSE [KWAIT	
38.	<predefined_port>	=	KCTL [KCOM1	
39.	<valid_bit>	=	[2 [4 [8 [16 [32	
40.	<iterations>	=	<integer>	[TINTEGER]
41.	<integer>	=	TINTEGER	
42.	<identifier>	=	TIDENTIFIER	
43.	<string>	=	TSTRING	
44.	<open_parenthesis>	=	TLPAREN	
45.	<close_parenthesis>	=	TRPAREN	
46.	<period>	:=	TDOT	

APPENDIX E: Example CPL Source Program

```
* Port declarations
Ports
    PortC 64259 Output:
    PortA 64256 Input:
    Com1port COM1: 300 7 2 0:          * Serial port declaration
End

* Device declarations
Devices
    PalletLiftUp      Pulse          PortC    4:
    Conveyor          Coil           PortC    5:
    PhotoCell         Sensor         PortA    7:
    PalletArrived     Sensor         PortA    6:
    ChuckOpen         Pulse          PortC    1:
    LatheG66inp       Pulse          PortC    0:
    Robot             Programmable   LPT1:
    Lathe             Programmable   Com1port:
    LatheStart        Pulse          PortC    2:
    LatheStop         Sensor         PortA    4:
    PalletLifted      Sensor         PortA    5:
    PalletStops       Coil           PortC    0:
    ChuckClose        Pulse          PortC    3:
    PalletLiftDown    Pulse          PortC    6:
    LatheRunning      Sensor         PortA    2:
    LatheHandShk      Pulse          PortC    3:
    Delay             Wait:
End
```

```
* Procedure operations
Procedure
    LatheHandShk.Strobe;
    LatheG66inp.Strobe;
    Lathe.Do(loadlathe);
    Robot.Send("NT");
    PalletStops.On;
    Conveyor.On;
    PhotoCell.WaitOn;
    PalletStops.Off;
    PalletArrived.WaitOn;
    Delay.1000;
    PalletLiftUp.Strobe;
    PalletLifted.WaitOn;
    Conveyor.Off;
    ChuckOpen.Strobe;
    Robot.Do(Loadpart);
    Delay.1000;
    ChuckClose.Strobe;
    Delay.2000;
    Robot.Do(Moveaway);
    Delay.2000;
    LatheStart.Strobe;
    LatheStop.WaitOff;
    Robot.Do(Moveback);
    Delay.2000;
    ChuckOpen.Strobe;
    Delay.2000;
    Robot.Do(Getpart);
```

```
    PalletStops.On;
    PalletLiftDown.Strobe;
    Conveyor.On;
    Delay.500;
    Conveyor.Off;
    PalletStops.Off;
    LatheStart.Strobe;
    LatheHandShk.Strobe;
End;
```

APPENDIX F: CPL Compiler Source Code

```
//------------------------------[ main.cpp ]---------------------------------
/*  ************************************************************************ *
/
/*  This is the main program of the CPL compiler.  This program is copyrighte
d
    and belongs to the Applied Science Division of Miami University.  Copying
    source code is illegal.
*/
/* NAME:         Main
         AUTHOR:  Meghamala Nugehally
         DATE:    10/16/92
         VERSION:        4.0
*/
/*  ************************************************************************ */

#include <stdio.h>
#include "..\head\errors.h"
#include "..\head\trans.h"
#include "..\head\table.h"
#include "..\head\list.h"
#include "..\head\crossref.h"

table symbol_table(TBLSZ);                        // symbol table of known size
table* sytb = &symbol_table;
list_t* procPtr;                 // list of executable statements
error_t errhandler("..\\prj\\errors.dat");
error_t* ee = &errhandler;

int numports = 0;          // to be eliminated as global
int numtypes = 0;          // to be eliminated as global

/*  ************************************************************************ *
/
/* FUNCTION:    The main program instantiates the program as a translate
   object and calls the parsing functions to begin the compilation process.
   After the completion of parsing, crossreferences are built.
*/
/*  ************************************************************************ */

main(int argc, char* argv[])
{
    // if file name not entered, prompt user and get filename
         if (argc != 2)
                printf("Enter program file name: %s\n", gets(argv[1]));

                translate_t program(argv[1]);

      program.parsePorts();          // parse ports section
                program.parseDevices();        // parse device section
                program.parseProcedure();      // parse procedure section
                program.generate();            // Generate code
                program.crossrefer();          // Generate cross reference lis
ting
                if (ee->Errors() == 0)
                {
        printf("\n Compilation successfully completed\n");

                }
                else
                {
```

```
                        printf("Compilation completed with %d errors\n", ee->
Errors());
                }

}

/* end of file */
```

```
//-------------------------------[ char.h ]---------------------------------
~

#ifndef CHAR_H
#define CHAR_H

#define LINELEN 80

#include <stdio.h>

// The character class identifies different categories of characters in the
// ascii set. It reads characters from the source file and returns them to
// the scanner a character at a time.

class char_t {
    int reget_flag;                    // to provide for one character look
ahead
    int last_char;          // the character read from the file
    int nextchar;
    FILE *fp;
         char* nextline, *thisline;
public:
    char_t(FILE *f) { reget_flag = 0; last_char = 0; fp = f;
                                  nextchar = 0;
                                  nextline = new char[LINELEN];
                                  thisline = new char[LINELEN];
                                  Readline(); }
    void reget(void) { reget_flag = 1; };
    int code(void) { return(last_char); };
    int next(void);
    void Readline();
    void reset_nextchar() { nextchar = 0; }
         short Class(void);
         char* Nextline() { return (nextline); }
         char* Thisline() { return (thisline); }
};

/* Character Classes */

#define CILL    0       /* illegal character */
#define CWHITE  1       /* white space */
#define CQUOTE  2
#define CID     3       /* ok for identifiers */
#define CLPAREN 4
#define CRPAREN 5
#define CCOMMA  6
#define CDOT    7
#define CDIG    8
#define CCOLON  9
#define CSEMI   10
#define CLF     11      /* line feed */
#define CEOF    12      /* end of file */
#define CCOMMNT 13              /* comment character */


#endif
```

```
//--------------------------[ char.cpp ]-----------------------------------

/* This file initializes a table of valid character constants, and defines
        the member functions for the char class
*/

#include <string.h>
#include "..\head\char.h"
#include "..\head\errors.h"

extern error_t * ee;       // error object declared in main.cpp

short ch_class[] = {
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,          /* Nu,...*/
 CILL,CWHITE,CLF,CWHITE,CWHITE,CWHITE,CILL,CILL,   /* bs,ht,lf,vt,ff,cr,so,si*
/
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,
 CWHITE,CILL,CQUOTE,CID,CID,CID,CILL,CILL,         /* sp,!,",#,$,%,&,'*/
 CLPAREN,CRPAREN,CILL,CILL,CCOMMA,CILL,CDOT,CILL,  /* (,),*,+,,,-,.,/ */
 CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,          /* 0,1,2,3,4,5,6,7 */
 CDIG,CDIG,CCOLON,CSEMI,CILL,CILL,CILL,CILL,       /* 8,9,:,;,<,=,>,? */
 CILL,CID,CID,CID,CID,CID,CID,CID,                 /* @,A,B,C,D,E,F,G */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* H,I,J,K,L,M,N,O */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* P,Q,R,S,T,U,V,W */
 CID,CID,CID,CILL,CILL,CILL,CILL,CID,              /* X,Y,Z,[,\,],^,_ */
 CILL,CID,CID,CID,CID,CID,CID,CID,                 /* `,a,b,c,d,e,f,g */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* h,i,j,k,l,m,n,o */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* p,q,r,s,t,u,v,w */
 CID,CID,CID,CILL,CILL,CILL,CILL,CILL              /* x,y,z,{,|,},~,del */
};

/*  ********************************************************************** *
/
/*
        FUNCTION:       Reads one source code line from the cpl program file
                                        Comments are ignored, and single char
acter backtracking is
                implemented.
*/
/*  ********************************************************************** *
/

void char_t::Readline()
{
        strcpy(thisline, nextline);      // make a copy of the current line
        if (fgets(nextline, LINELEN+1, fp) == NULL)        // read next line
        {       printf("%-80s", ee->readerror(_F_EREADSRCFIL));
                ee->checkerrors(FATAL_T);
        }

}
/*  ********************************************************************** */
/*
   FUNCTION:    If the reget flag is set, returns the previous character read
   else, reads the next character from the source file and returns it.
*/

/*  ********************************************************************** */
int char_t::next(void)
```

```
{
        if (reget_flag)
        {
                reget_flag = 0;     // Reset reget flag if set
        } else
        {
                if (nextline[nextchar] == '*')      // check for comment chara
cter
      {
                        last_char = '\n';
                        nextchar = 0;
                }
                else
                {       last_char = nextline[nextchar];
                        nextchar++;
      }
        }
        return(last_char);     // Return previous character if reget flag set
}
/*  ********************************************************************** *
/
/*
        FUNCTION:    Returns a unique character code that identifies the char
acter
*/
/*  ********************************************************************** *
/

// If the character read is not end-of-file, then return the character type,
// else return the end-of-file code.

short char_t::Class(void) { return((last_char != EOF) ?
                                                                        ch_c
lass[last_char] : CEOF); }

/*  ********************************************************************** *
/

/* end of file */
```

```
//-----------------------------[crossref.h]---------------------------------

#ifndef CROSSREF_H
#define CROSSREF_H

#include <stdio.h>

#define MAX_ID          30
#define MAXLEN          10
#define MAXSTR          80
#define SPACE_0 ""
#define SPACE_19        "                   "


typedef int ndxarr[10];

// device cross reference record
struct devarry
{
        char deviceName[MAX_ID+1];
        char devType[MAX_ID+1];
        int     bit;
        char outport[MAX_ID+1];
};

// port cross reference record
struct declarry
{
        char deviceName[MAX_ID+1];
        char portName[MAX_ID+1];
        int bit;
   char outport[MAX_ID+1];
};

// port cross reference list
struct crf_1
{
        char portname[MAX_ID+1];
        devarry devinfo[10];         // devices associated with a port variab
le
};

// device cross reference list
struct crf_2
{
        char typename[MAX_ID+1];
        declarry devtp[10];      // device variables associated with a device
 type
};

//cross reference class to build the cross reference table and print it
class crossref_t
{
        crf_1 ptod[10];
        crf_2 ttod[10];
        FILE *crp;

public:
        void build(ndxarr, ndxarr);
        void print(char*, ndxarr, ndxarr);
```

```
        void display(char* );
};
#endif
```

```
// --------------------------crossref.cpp---------------------------------
#include "..\head\crossref.h"
#include "..\head\table.h"
#include "..\head\errors.h"

extern table* sytb;
extern int numports;
extern int numtypes;
extern error_t *ee;


/*  ********************************************************************** *
/

/*
        FUNCTION:    Builds a cross reference between port variables and devi
ce
        variables.  For each port variable, a list of the devices that use th
e port
        including the device name, device types and bit number is created. Pr
ovides
        useful debugging information.

*/

/* ********************************************************************** *
/

void crossref_t::build(ndxarr ind1, ndxarr ind2)
{
        symbol_t* ss;
        int i = 0;              // index of the latest  port name added to pt
od
        int z=0;        // index of the latest type name added to ttod
        int j = 0;      // index of existing port name if matches current port
 name
        int  k, y;      // k = number of unique ports, y = number of unique ty
pes
        int x = 0;      // index of existing type name if matches current type
 name
        unsigned m=0;  // counter for # of devices for which cross reference
is built
   unsigned l=0;  // index for symbol table
        char portt[MAX_ID+1]; // temporary variable to hold  port name
        char typet[MAX_ID+1]; // temporary variable to hold  type name

        // Build the port and device cross reference
        // Get devices from the symbol table to build the cross
        // reference list

        while (((ss = sytb->getsymbol(l)) != NULL) && m<sytb->get_numOfdev())
        {
                if (ss->type == KDEVICES &&
                        ((((device_t*) (ss->symbol))->get_devType()) != KWAIT
))
                {
                        strcpy(typet,((device_t*) (ss->symbol))->get_typename
());
                        switch (((device_t*)(ss->symbol))->get_devType())
                        {
```

```
                                case KCOIL:
                                        strcpy(portt,((coil_t*)(((device_t*)
(ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KSENSOR:
                                        strcpy(portt,((sensor_t*)(((device_t*
) (ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KPULSE:
                                        strcpy(portt,((pulse_t*)(((device_t*)
 (ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KPROGRAMMABLE:
                                        strcpy(portt,((programmable_t*)(((dev
ice_t*) (ss->symbol))->getDevptr()))->get_comportName());
                                        break;
                                default:
                                        break;
                        }
                        j=0;x=0;

                    // check if port variable already entered in list
                        while ((strcmp(portt,ptod[j].portname) != 0) && j<i)
j++;
          // check if type name already entered in list
                        while ((strcmp(typet,ttod[x].typename) != 0) && x<z)
x++;
                        // if port variable already in list, add device to
                        // corresponding port row
                        if (j<i)
                        {
                                k=ind1[j]+1;
                        }
                        // else add the port variable and the corresponding
                        // device variable to a new row.
                        else
                        {
                                strcpy(ptod[i].portname,portt);
                                k=0;i++;
                        }

          // do the same for the device cross reference list
                        if (x<z)
                                y = ind2[x]+1;
                        else
                        {
                                strcpy(ttod[z].typename,typet);
                                y=0;z++;
                        }
          // copy device name
                        strcpy(ptod[j].devinfo[k].deviceName,
                                        ((device_t*)(ss->symbol))->get_devic
eName());
                        strcpy(ptod[j].devinfo[k].devType,
                                        ((device_t*)(ss->symbol))->get_typen
ame());
                        strcpy(ttod[x].devtp[y].deviceName,
                                        ((device_t*)(ss->symbol))->get_devic
eName());
                        strcpy(ttod[x].devtp[y].portName,portt);
```

```
                        ind1[j] = k;  // # devices that use jth  port variabl
e is k
                        ind2[x] = y;  // # devices that are of xth type is y

                        // copy remaining information about devices
                        switch (((device_t*)(ss->symbol))->get_devType())
                        {
                                case KPROGRAMMABLE:
                                        strcpy(ttod[x].devtp[y].outport,
                                                ((programmable_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_comportName());
                                        strcpy(ptod[j].devinfo[k].outport,
                                                ((programmable_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_comportName());
                                        break;
                                case KCOIL:
                                        ptod[j].devinfo[k].bit = ((coil_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((coil_t*)(((d
evice_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                case KSENSOR:
                                        ptod[j].devinfo[k].bit = ((sensor_t*)
(((device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((sensor_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                case KPULSE:
                                        ptod[j].devinfo[k].bit = ((pulse_t*)(
((device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((pulse_t*)(((
device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                default:
                                        break;
                        }
                        m++;l++;
#ifdef VERBOSE
                        printf("%s %s %s %d\n", ptod[j].portname,
                                        ptod[j].devinfo[k].deviceName,
                                        ptod[j].devinfo[k].devType,ptod[j].d
evinfo[k].bit);
#endif
                }
                else
                        l++;
        }
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Outputs the cross reference when the compilation is compl
ete
                                        to a seperate file that has the sourc
e file name but with an
                                        extentsion of .ref.
```

```
*/

/* ********************************************************************** *
/

void crossref_t::print(char* crfile, ndxarr p, ndxarr q)
{
        char space[20];       // number of spaces

        // create cross reference file with same filename as input file but w
ith
        // an extension of 'ref'
        if ((crp = fopen(crfile, "w")) == NULL)
        {       printf("%s\n", ee->readerror(_W_UNOPNCRFILE));
                ee->checkerrors(WARNING_T);
                // Error opening cross reference file\n");
   }

        else
        {
                fprintf(crp,"***********Cross references for ports and devic
es*********\n\n");

                for (int i=0; i <= numports; i++)
                {
                        fprintf(crp,"%-19s", ptod[i].portname);
                        strcpy(space, SPACE_0);       // contains a null stri
ng for the first line
                        for (int j=0; j<=p[i]; j++)
                        {
                                if (strcmp(ptod[i].devinfo[j].devType, "PROGR
AMMABLE") == 0)
                                        fprintf(crp,"%s %-14s %-15s\n", space
, ptod[i].devinfo[j].deviceName,
                                                ptod[i].devinfo[j].devType);
                                else
                                        fprintf(crp,"%s %-14s %-14s %d\n", sp
ace, ptod[i].devinfo[j].deviceName,
                                                ptod[i].devinfo[j].devType,
ptod[i].devinfo[j].bit);
                                strcpy(space, SPACE_19);  // contains 19 spac
es for all other lines

                        }
                }

                fprintf(crp,"\n***********Cross references for device types
and devices*********\n\n");

                for (int x=0; x<=numtypes; x++)
                {
                        fprintf(crp,"%-19s", ttod[x].typename);
                        strcpy(space, SPACE_0);    // contains  a null string
 for the first line

                        for (int y=0; y<=q[x]; y++)
                        {
                                if (strcmp(ttod[x].typename,"PROGRAMMABLE") =
= 0)
```

```
                                        fprintf(crp,"%s %-14s %-15s\n", space
, ttod[x].devtp[y].deviceName,
                                                ttod[x].devtp[y].portName);
                                else
                                        fprintf(crp,"%s %-14s %-14s %d\n", sp
ace, ttod[x].devtp[y].deviceName,
                                                ttod[x].devtp[y].portName, t
tod[x].devtp[y].bit);
                                strcpy(space, SPACE_19);    // contains 19 spa
ces for all other lines

                        }

                }
                fclose(crp);
        }
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Displays the cross reference listing on the screen at the
 end
                                        of compilation.

*/

/* ********************************************************************** *
/

void crossref_t::display(char* fn)
{
   char refstr[MAXSTR];

        if ((crp = fopen(fn, "r")) != NULL)
        {
                while (!feof(crp))
                {
                        if (fgets(refstr, MAXSTR, crp) != NULL)
                                printf("%s", refstr );
                }
                fclose(crp);
        }
        else {
                printf("%s\n", ee->readerror(_W_UNOPNCRFILE));
                ee->checkerrors(WARNING_T);
   }
}

/* end of file */
```

```
//---------------------------------errors.h-------------------------------

#ifndef ERRORS_H
#define ERRORS_H

#include <stdio.h>

#define codestring(errcode)      (#errcode)

#define MXLEN                              80
#define ERROR_T                 0
#define WARNING_T               1
#define FATAL_T                 2
#define _F_UNOPNSRCFIL  100
#define _F_UNOPNERRFIL  101
#define _F_UNOPNOUTFIL  102
#define _F_EREADSRCFIL 103
#define _F_PARSERERROR 104
#define _F_UNOPNCMDFIL 105
#define _F_UNREAERRFIL 106
#define _F_UNRESERRPTR 107
#define _E_SEMICOLEXPT  111
#define _E_FUNCOPREXPT  112
#define _E_UNDEFIDENTF 113
#define  _E_IDENTIFEXPT 114
#define  _E_TYPADDREXPT 115
#define _E_INCORSERPOR 116
#define _E_INTEGEREXPT 117
#define _E_INVALDEVTYP 118
#define _E_DEVTYPEXPTD 119
#define _E_INVALIDFUNC 120
#define _E_KEYWORDEXPT 121
#define _E_PARTYPMISMA 122
#define _E_PORTKEYEXPT  123
#define _E_DEVKEYWEXPT 124
#define _E_PROCKEYEXPT 125
#define _W_UNOPNCRFILE 150
#define _W_BAUDNOTSPEC  151
#define _W_STMTNOEFFEC 152
#define _W_UNOPNTRCFIL 153

class error_t {
        int err_type;           // FATAL/ERROR/WARNING
        int err_count;          // number of syntax errors
        int warn_count; // number of warning errors
        int tot_count;
        int err_limit;          // condition to terminate compilation
        int err_code;           // error code
        char* err_msg;          // error message
        FILE * ep;                      // error data file pointer
public:
        error_t(void);
        error_t(char* );
        char* readerror(int );
        void checkerrors(int);
        int Errcount() { return (err_count); }
        int Warncount() { return (warn_count); }
        int Errors() { return (tot_count); }
};
```

```
#endif
```

```
//------------------------------ errors.cpp ---------------------------------

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "..\head\errors.h"

/* ************************************************************************ *
/
/*
                FUNCTION:       The error class constructor initializes the e
rror object and
                allocates storage for it.  It opens the errors file errors.da
t.
/*
/* ************************************************************************ *
/

error_t::error_t(char* fname)
{
        if ((ep = fopen(fname, "r")) == NULL)
   {
                printf(" Unable to locate errors file "errors.dat\n");
                printf ("Copy "errors.dat" into the "prj" directory and recom
pile\n");
        }
        else
   {
                err_count = 0;
                warn_count = 0;
                tot_count = 0;
                err_limit = 5;
        }
}

/* ************************************************************************ *
/
/*
                FUNCTION:       This function retrieves error messages from t
he error file.
                The error message is matched with an error code which is pass
ed to the
      function by the calling routine.
/*
/* ************************************************************************ *
/

char* error_t::readerror(int ec) {

        char errstr[MXLEN];
        char revstr[MXLEN];
        char * msgstr, *codestr;
        char* p;

        if (fseek(ep, 0L, SEEK_SET) == 0)
        {
                while (!feof(ep))
                {       if ((fgets(errstr, MXLEN, ep)) != NULL)
                        {    sscanf(errstr, "%d", &err_code);
```

```
                                if (err_code == ec)
                                {       strrev(errstr);
                                        p = strrchr(errstr, ' ');
                                        *p = '\0';
                                        strrev(errstr);
                                        return (errstr);

                                }
                        }
                        else
                        {
                                printf("%s\n", readerror(_F_UNREAERRFIL));
                                checkerrors(FATAL_T);
         }
                }  // end of while
        }
        else
        {
                printf("%s\n", readerror(_F_UNRESERRPTR));
                checkerrors(FATAL_T);
        }
        return (NULL);
}

/* ********************************************************************** *
/
/*
                FUNCTION:       This function retrieves error messages from t
he error file.
                The error message is matched with an error code which is pass
ed to the
      function by the calling routine.
/*
/* ********************************************************************** *
/

void error_t::checkerrors(int ty)
{
        switch (ty)
        {
                case ERROR_T:
                        err_count++;
                        tot_count = err_count + warn_count;
                        if (tot_count > err_limit)
                        {       printf ("Compilation ended with %d errors\n",
 tot_count);
                                exit(-1);
                        }
                        break;
                case WARNING_T:
                        warn_count++;
                        tot_count = err_count + warn_count;
                        if (tot_count > err_limit)
                        {       printf ("Compilation ended with %d errors\n",
 tot_count);
                                exit(-1);
                        }
                        break;
                case FATAL_T:
                        printf("Fatal error discovered\n");
```

```
                        printf("Compilation cannot continue\n");
                        printf("%d errors discovered during compilation\n", t
ot_count);
                        exit(-2);
                        break;
                default:
                        break;
        }
}
```

```
// --------------------------FUNCTBL.H--------------------------------

#ifndef FUNC_H
#define FUNC_H

#define DEVINDEX                5
#define FUNCINDEX               10
#define MAXDEV                  5
#define MAXFUNC         8
#define MAXPARAM                10
#define CVALID                  1
#define CINVALID                0
#define NOPAR                   0
#define NA                              0

/* Function opcode definitions */

#define CON                             1
#define COFF                    2
#define CWAITON         3
#define CWAITOFF                4
#define CSEND                   5
#define CWAIT                   6
#define CSTROBE         7
#define CDO                             8
#define CSERPORT                9
#define CPORT                   10
#define CSOURCE         11

// define the opcode and parameters for a function and indicate if valid
// for a given device type
typedef struct {
        int valid;
        int opcode;
        int plist[MAXPARAM];     // list of parameter types
}funcInfo;

// define all functions for all device types
typedef funcInfo tbltype[MAXDEV][MAXFUNC];

#endif
```

```
// -------------------------[functbl.cpp]--------------------------------

#include "..\head\functbl.h"
#include "..\head\token.h"

/* Initialize function table with valid functions and parameters */
tbltype functbl =

/* coil device */              {  {
/* func ON     */                         { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func off    */                         { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func waiton */                         { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func waitoff*/                         { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func send    */                        { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func wait    */                        { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func strobe  */                        { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func do      */                        { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                  },
/* sensor device */               {
                                          { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                  },
/* pulse device  */               {
                                          { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CVALID, CSTROBE, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                          { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
```

```
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                        },
/* for programmable */                  {
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CVALID, CSEND, {TCHARSTRING, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CVALID, CDO, {TIDENTIFIER, NOPAR, N
OPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                        },
/* Plain functions */                   {
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CVALID, CWAIT, {TINTEGER, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                                { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                        }
                              };
/* END OF FILE */
```

```
//--------------------------------[ list.h ]----------------------------------
-
/* This is a generic list class -- change the type names and reuse.  */

#ifndef LIST_H
#define LIST_H

#include "..\head\stmts.h"

//  Change this typedef to make the list operate on other types
typedef procedure_t * objptr_t;

struct entry_t  {                  // doubly linked list entry
    objptr_t obj;
    struct entry_t *prev, *next;
    };

//  class definition for a list of "statement" objects.
class list_t
{

    public:
        list_t(void);
        ~list_t(void);

        void insert(objptr_t obj);
        void append(objptr_t obj);
        void remove(objptr_t obj);
        int  length(void);

        objptr_t first(void);
        objptr_t next(void);
        objptr_t last(void);
        objptr_t prev(void);

   private:
        entry_t head, tail, *cursor;
        int n_entries;

};


#endif

// end of file
```

```
//--------------------------------[ list.cpp ]--------------------------------
-

/* AUTHOR:           Charles Ames
        MAINTAINED BY:          Meghamala Nugehally
        DATE:                   02/20/92

        Member functions for the list class.  This list class has the
        property that elements are always added at the head.

*/


#include <stdio.h>
#include "..\head\list.h"

/*  ********************************************************************** *
/

/*
   FUNCTION:    This is the constructor function for the list class. It creat
es
   an empty list and initializes the head and tail pointers to null.
*/

/*  ********************************************************************** *
/
list_t::list_t(void)
{

        head.obj = ( objptr_t )NULL;
        head.next = &tail;
        head.prev = (entry_t *)NULL;

        tail.obj = ( objptr_t )NULL;
        tail.prev = &head;
        tail.next = (entry_t *)NULL;

        cursor = &head;
        n_entries = 0;

        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    This is the destructor function for the list class. It
   deallocates memory for the current list object.
*/
/*  ********************************************************************** *
/

list_t::~list_t(void)
{
        entry_t *temp;
        cursor = head.next;

        while (cursor != &tail)
        {
```

```
                temp = cursor;
                cursor = cursor->next;
                delete temp;
        }
        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    It inserts a list element at the head of the list and updates
   the length of the list.
*/
/*  ********************************************************************** *
/

void list_t::insert( objptr_t obj)
{
        cursor = head.next;
        head.next = new entry_t;
        head.next->obj = obj;
        head.next->next = cursor;
        head.next->prev = &head;
        cursor->prev = head.next;

        n_entries++;

        return;
}


/*  ********************************************************************** *
/

/*
   FUNCTION:    It inserts a list element at the tail of the list and updates
   the length of the list.
*/
/* ********************************************************************** *
/

void list_t::append(objptr_t obj)
{

        cursor = tail.prev;
        tail.prev = new entry_t;
        tail.prev->obj = obj;
        tail.prev->prev = cursor;
        tail.prev->next = &tail;
        cursor->next = tail.prev;

        n_entries++;

        return;
}

/*  ********************************************************************** *
/

/*
```

```
   FUNCTION:    It deletes the list element that matches the element passed t
o
   it as input parameter, and updates the length of the list.
*/
/* ********************************************************************** *
/

void list_t::remove( objptr_t obj)
{
        int deleted = 0;

        cursor = head.next;

        while (cursor != &tail && !deleted)
        {
                if (cursor->obj == obj)
                {
                        cursor->prev->next = cursor->next;
                        cursor->next->prev = cursor->prev;
                        cursor->obj = ( objptr_t )NULL;
                        delete cursor;
                        deleted = 1;
                        n_entries--;
                }
                else
                {
                        cursor = cursor->next;
                }

        }

        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element at the head of the list.
*/
/* ********************************************************************** *
/

objptr_t list_t::first(void)
{

    cursor = head.next;
         return cursor->obj;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element next to the current.
*/
/* ********************************************************************** *
/

objptr_t list_t::next(void)
{
         if (cursor != &tail) cursor = cursor->next;
```

```
        return cursor->obj;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element at the tail of the list.
*/
/* ********************************************************************** *
/

objptr_t list_t::last(void)
{
        cursor = tail.prev;
        return cursor->obj;

}

/*  ********************************************************************** *
/
/*
       FUNCTION:    Returns the element before the current.
*/
/* ********************************************************************** *
/

objptr_t list_t::prev(void)
{
        if (cursor != &head) cursor = cursor->prev;
        return cursor->obj;

}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the length of the list.
*/
/* ********************************************************************** *
/

int list_t::length(void)
{
        return n_entries;

}
// end of file
```

TRANSLATOR BASE STATIC

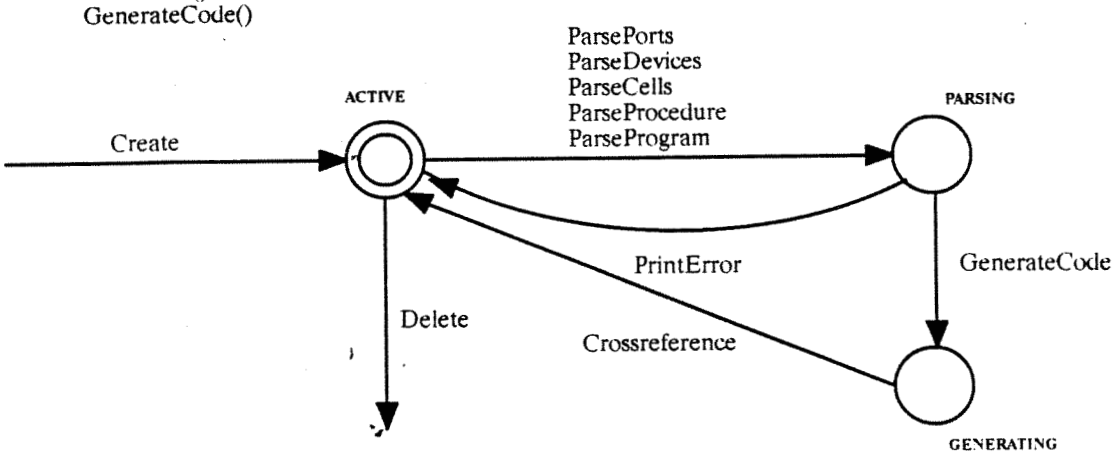
The main class of the translating system. Initializes the variables, open files, and initiates the parsing process.

Attributes

ProgramFile
Token
Character
Source Directory
Output Directory

Interface

Create()
Delete()
ParsePorts()
ParseDevices()
ParseCells()
ParseProcedure()
ParseProgram()
PrintError()
GenerateCode()



SYMBOLTABLE BASE STATIC

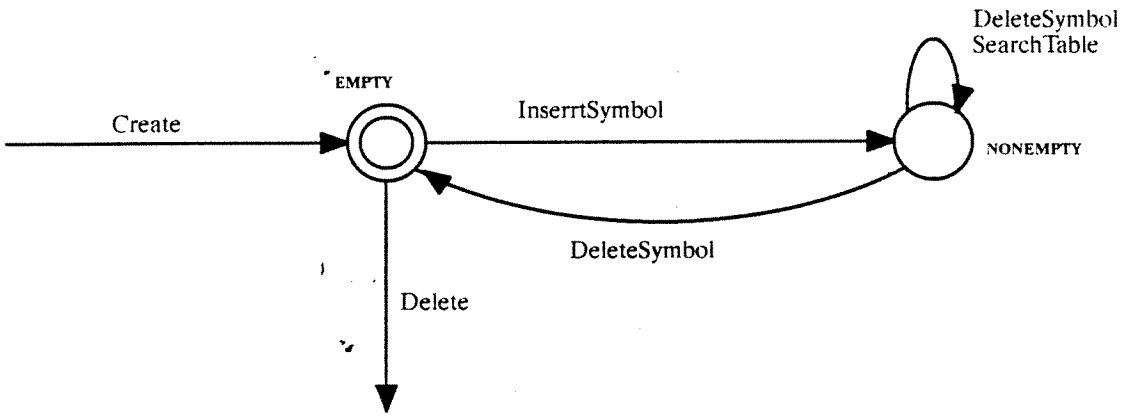
A table for storing information about port. device. and cell variables. The two mjor operations of the table are insert and search. Insert allows you to add a symbol into the table . and search allows you to search the table for the given symbol information.

Attributes

Symbol
Symbol type
Pointer to next symbol

Interface

Create()
Delete()
InsertSymbol)
ReturnSymbol)
ConvertSymbolPointer()
SearchTable
DeleteSymbol()



FUNCTIONTABLE BASE STATIC

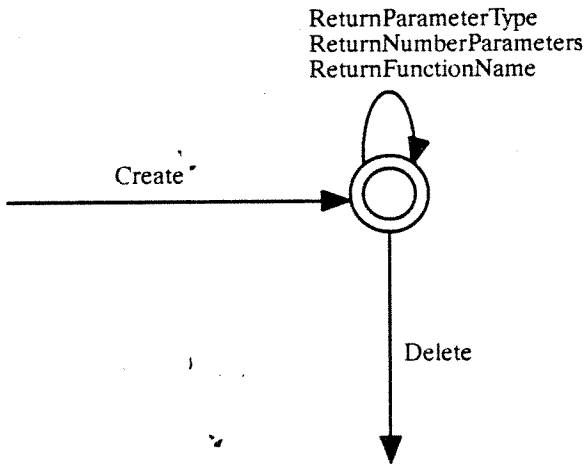
The function table is used to store information about CPL predefined functions. The number of parameters. the opcode. and the types of the parametrs are stored for each function.

Attributes

FunctionName
Number of Parameters
Parameter List
Opcode

Interface

Create()
Delete()
ReturnParameterType()
ReturnNumberParameters()
ReturnFunctionName()



CROSSREFERENCE BASE STATIC

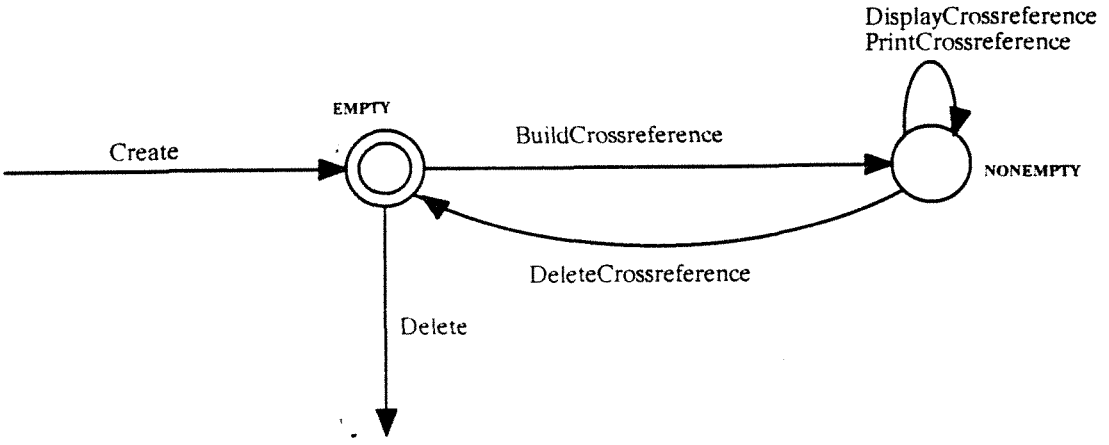
A class to put together all the crossreference information. This includes a list of the ports and the devices using each of the ports including the type of the device and the bit assigned for each port variable. A crossreference is created for the type of the device and thedevice variables declared to be of that type.

Attributes

PortCrossreference
DeviceCrossreference

Interface

Create()
Delete()
BuildCrossreference()
DisplayCrossreference()
PrintCrossreference()
DeleteCrossreference()



STATEMENT BASE STATIC

This generic class represents a statement as an object. This class will not be instantiated and only serves as a base class for specific statement types such as port statement, decive statement, etc., and hence there will not be any state transition diagram for it.

Attributes

StatementType

Interface

Create()
Delete()

TOKEN BASE STATIC

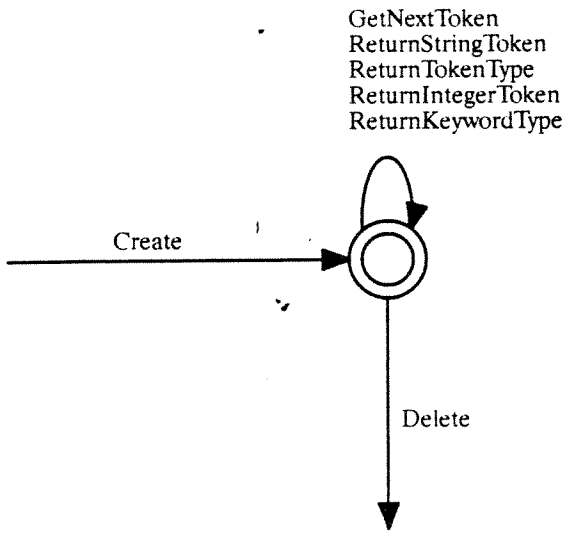
A token is a set of characters strung together. This class classifies tokens as integers, strings, keywords, special characters, illegal tokens, etc.

Attributes

Token Type
String Token
Integer Token
Keyword type

Interface

Create()
Delete()
GetNextToken()
ReturnStringToken)
ReturnTokenType()
ReturnIntegerToken()
ReturnKeywordType()



CHARACTER BASE STATIC

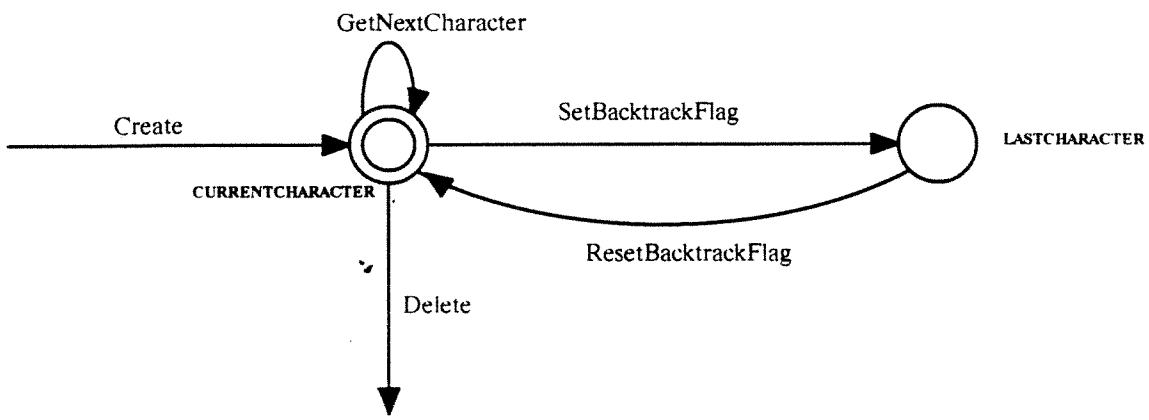
This class represents a character and all the attributes and operations associated with a character. It serves to scan the source code file for the next character and implements single character backtracking.

Attributes

CharacterScanned
BacktrackFlag

Interface

Create()
Delete()
GetNextCharacter()
SetBacktrackFlag()
ResetBacktrackFlag()



ERROR BASE STATIC

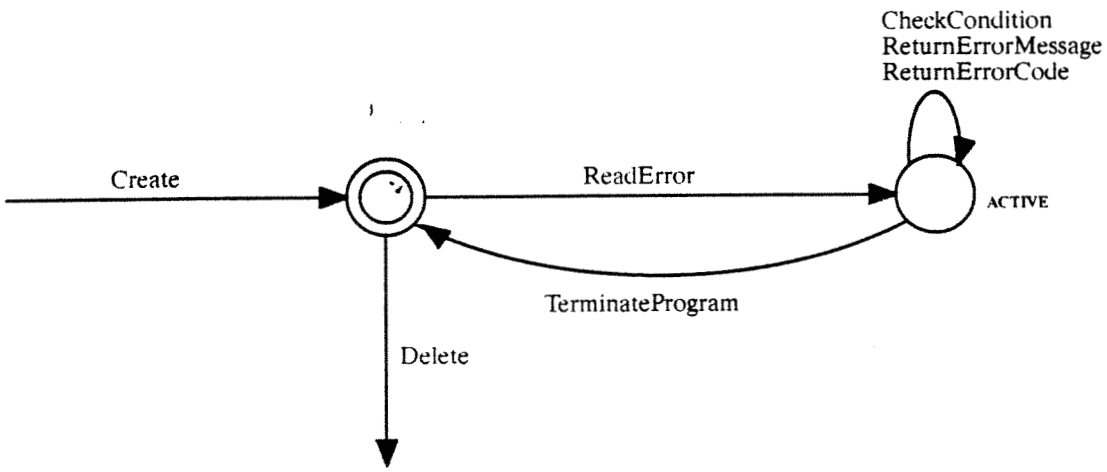
Represents the error database and is responsible for handling compile time errors. Contains an error database consisting of error codes and their corresponding messages. The error object reads errors from the database based on the error code passed to it. It checks for the error condition and either terminates the program or increments the error count depending upon the number of errors discovered or when a fatal error is encountered.

Attributes

ErrorType
ErrorFile
Number of Errors
Number of FatalErrors
Number of Warning Errors
ConditionType
ErrorMessage
ErrorCode

Interface

Create()
Delete()
ReadError()
CheckCondition)
ReturnErrorMessage()
ReturnErrorCode()
TerminateProgram()



APPENDIX C: Class Diagrams

Class diagrams depict the relationships between different classes. The notation used in class diagrams consists of dashed irregular shapes connected by arrows. The type of the arrow between any two classes represents the type of the relationship between them. One can represent four basic kinds of relationships: Inheritance, Instantiation, Containing, and Using

The notation consists of the following symbols:

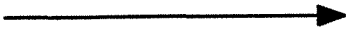
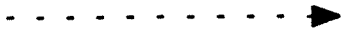
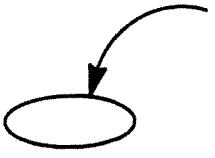
name	:	Label used to denote the name of the class
	:	Icon to denote a Class
	:	Inheritance Relationships– the class at the tail of the arrow is derived from or inherits the attributes and functions of the class at the head the arrow.
	:	Instantiation Relationship– the object at the tail of the arrow is an instance of the class at the head of the arrow
	:	Using Relationship– the class at the circle uses the services or functions of the class at the tail of the arrow in its implementation. The icon expands to a double line as it is made longer.
	:	Using Relationship– the class at the circle uses the services or functions of the class at the tail of the arrow in its interface.
	:	Containing Relationships– The class at the tail of the arrow contains the classes enclosed in the oval. The oval will have to be increased in size in order to accommodate the component classes.
	:	Icon to denote a class library
	:	Icon to denote a class inside a subsystem

Figure C.1 Using Relationships for Translator System

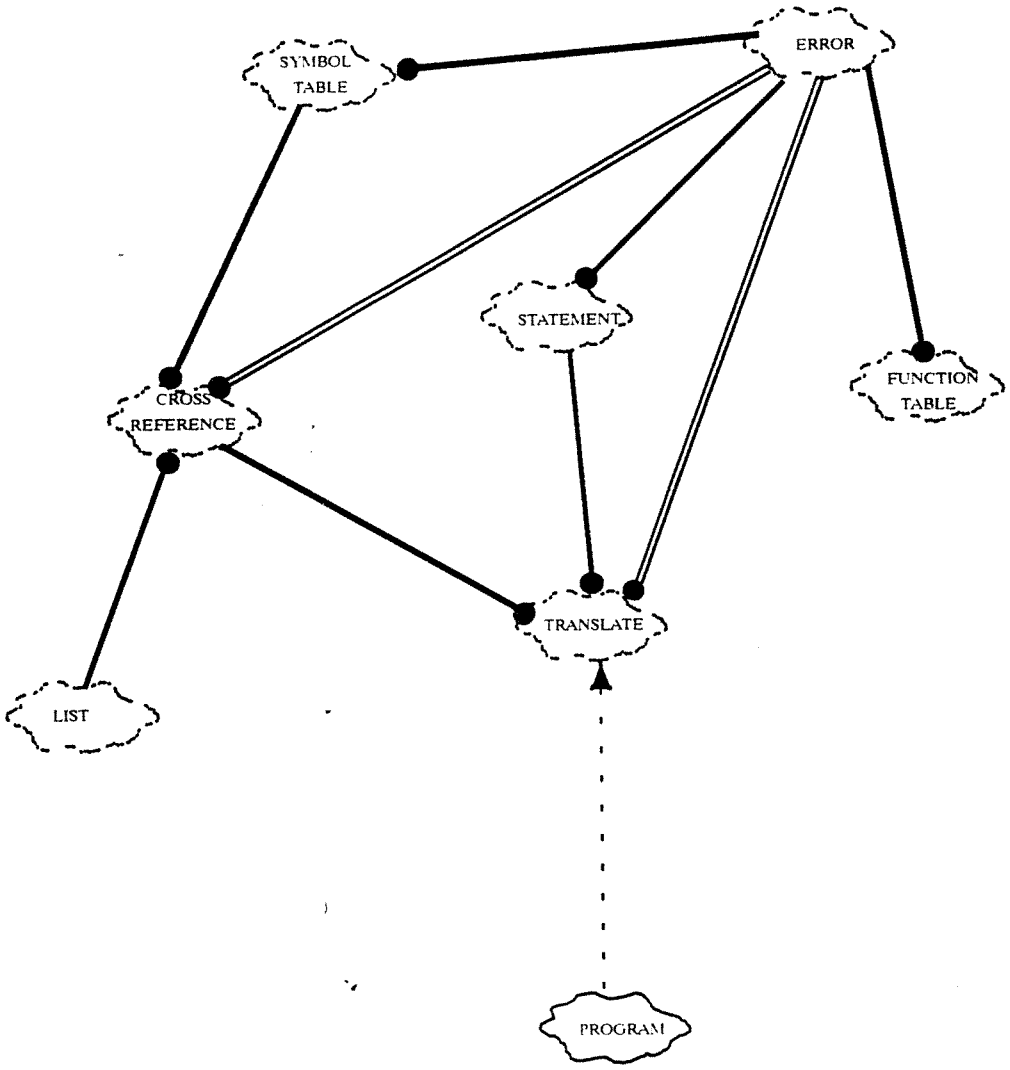


Figure C.2 Using Relationships for Translator System

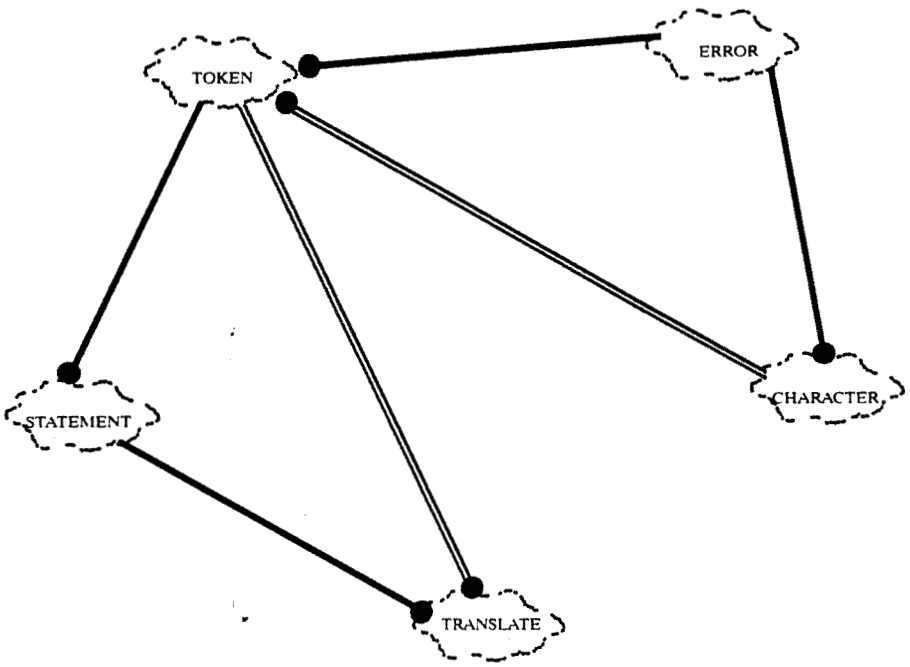


Figure C.3 Inheritance Relationships for Class Statement

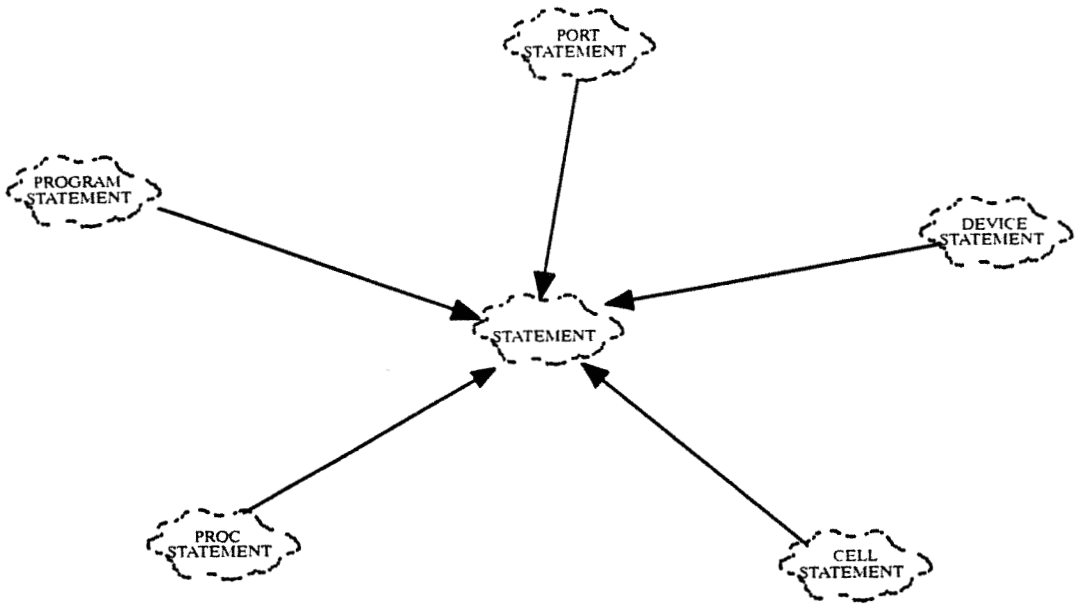


Figure C.4 Instantiation Relationships for Class Statement

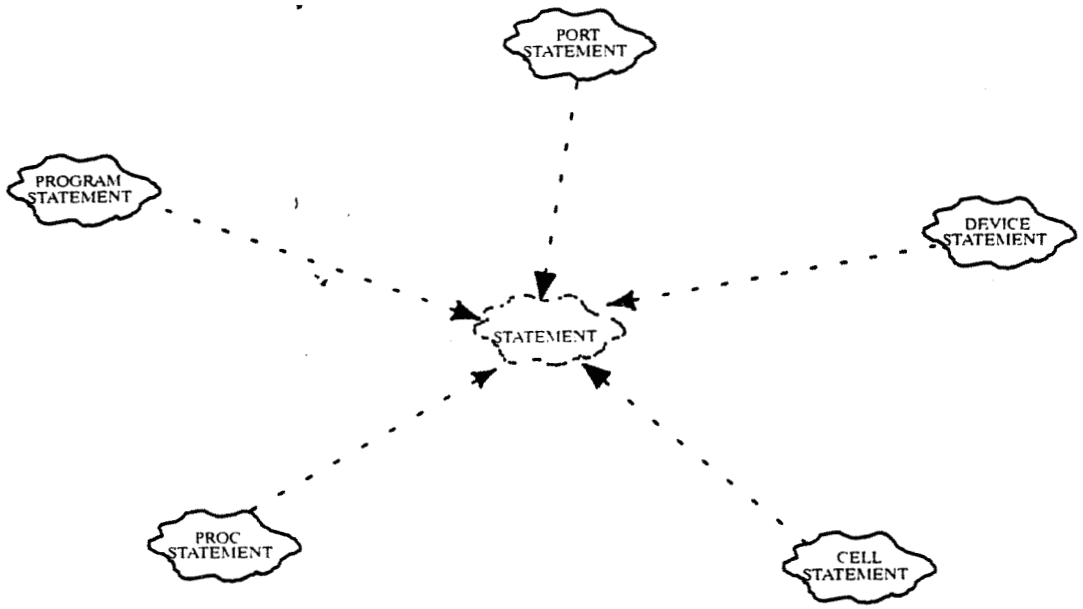


Figure C.5 Inheritance Relationships of Class Device Statement

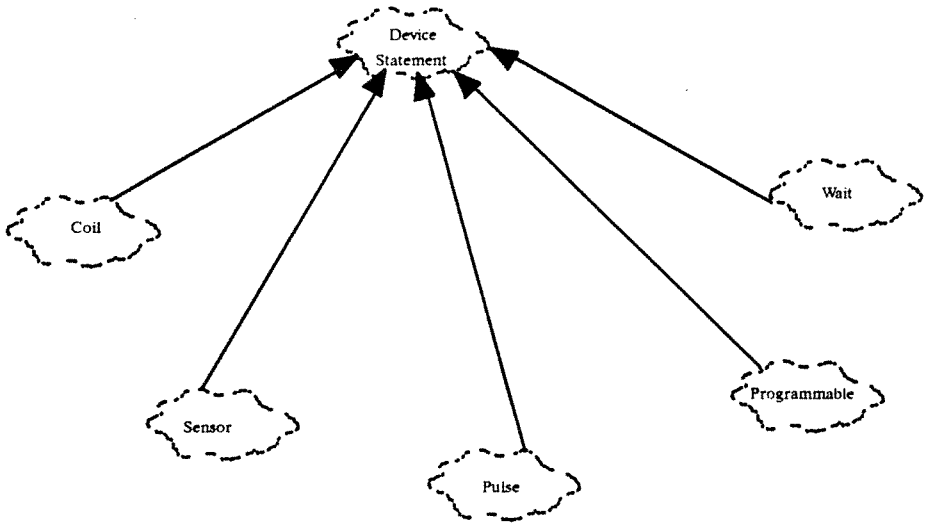
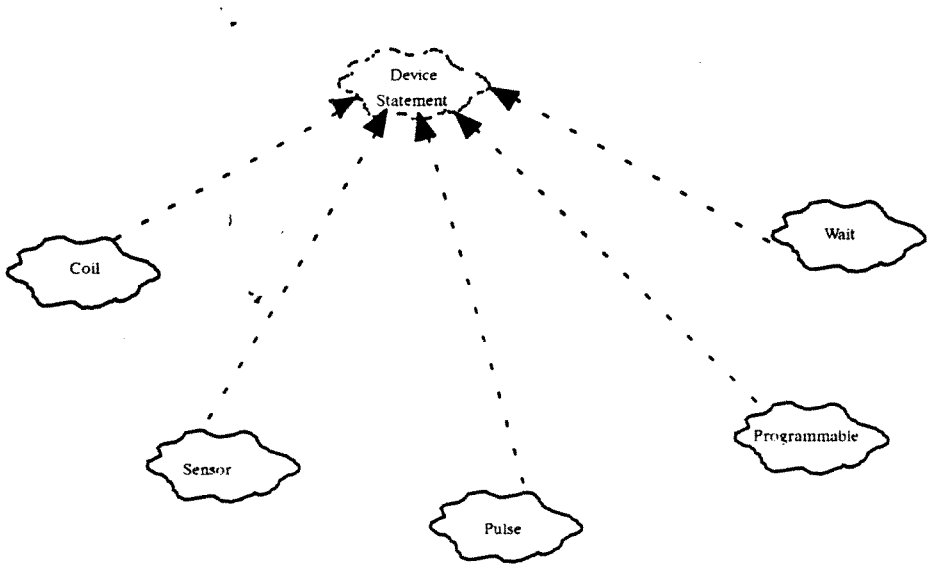


Figure C.6 Instantiation Relationships of Class Device Statement



APPENDIX D: CPL Grammar

The Cell Programming Language is designed as a context-free grammar and uses the Backus Naur notation for expressing the syntax. The grammar is free of ambiguity at present but shall be corrected for any ambiguities detected henceforth.

CPL is described as a set of production rules, each production rule consisting of a left hand side and a right-hand side, separated by an assignment operator. An example of a production rule is given below.

Production Rule : An example

<expr> -> <expr> + term> | <expr> - <term> | <term>

<term> -> 0|1|2|3|4|5|6|7|8|9

The left-hand side is always a non-terminal symbol. The right hand side may be a combination of non-terminals and terminals, only non-terminals, or only terminal symbols. The non-terminals are enclosed within the '<' and '>' symbols and the terminals are represented by constant values. The '—>' string serves as the assignment operator. Optional symbols may be enclosed within square brackets or braces '[' and ']'. Parenthesis are used to specify grouping of symbols, and when more than one symbol, separated by commas, is enclosed within '{' and '}', it means that at least one of the symbols or a group of symbols must be present. For example, in line 16 of the CPL grammar, the device_data must either consist of the port_name followed by a valid_bit or just a predefined_port. The parenthesis around port_name and valid_data indicates that these two are grouped and hence they go together. In this grammar, the language keywords, alphabets, special ASCII characters and digits are the terminal symbols. Currently, the language provides for sequence and repetition constructs, but also leaves room for adding alternative constructs. The CPL production rules and selection sets are shown in the following pages. Selection sets are sets of tokens that help determine the number of tokens necessary to apply the next production in a given rule. In this case, the a single token is sufficient to determine the next production to be applied. This makes the CPL grammar LL(1).

CPL Production Rules and Selection Sets

1.	<cpl_program>	:=	[<port_declarations>] <cell_declarations> <device_declarations> <procedure_declarations> <program_declarations>	[PORTS, DEVICES, CELLS, PROCEDURE, PROGRAM]
2.	<port_declarations>	:=	PORTS <port_stmtlist> END	[PORTS, LAMBDA]
3.	<cell_declarations>	:=	CELLS <cell_stmtlist> END	[CELLS]
4.	<device_declarations>	:=	DEVICES <device_stmtlist> END	[DEVICES]
5.	<procedure_declarations>	:=	PROCEDURE <procedure_name> <procedure_stmtlist> END	[PROCEDURE]
6.	<program_declarations>	:=	PROGRAM <program_stmtlist> END	[PROGRAM]
7.	<port_stmtlist>	:=	<port_stmt> [<port_stmtlist>]	[TIDENTIFIER, END]
8.	<port_stmt>	:=	<port_name> <port_address> <direction>	[TIDENTIFIER, TINTEGER, TKEYWORD]
		:=		
9.	<port_name>	:=	<identifier>	[TIDENTIFIER]
10.	<port_address>	:=	<integer>	[TINTEGER]
11.	<direction>	:=	[KINPUT] [KOUTPUT	
12.	<cell_stmtlist>	:=	<cell_stmt> [cell_stmtlist>]	[TIDENTIFIER, END]
13.	<cell_stmt>	:=	<cell_name> <cell_address>	[TIDENTIFIER, THOSTNAME, TNETADDRESS]
14.	<cell_name>	:=	<identifier>	[TIDENTIFIER]
15.	<cell_address>	:=	THOSTNAME [TNETADDRESS	
16.	<device_stmtlist>	:=	<device_stmt> [<device_stmtlist>]	
17.	<device_stmt>	:=	<device_name> <device_type> <device_data> TSEMICOLON	[TIDENTIFIER, TKEYWORD, TIDENTIFIER]
18.	<device_name>	:=	<identifier>	[TIDENTIFIER]
19.	<device_type>	:=	PROGRAMMABLE [<nonpgble_type>	[TKEYWORD, TIDENTIFIER]
20.	<device_data>	:=	[(<port_name> <valid_bit>), <predefined_port>]	[TIDENTIFER, TKEYWORD]
21.	<procedure_stmtlist>	:=	<procedure_stmt> [<procedure_stmtlist>]	[TIDENTIFIER, END]
22.	<procedure_stmt>	:=	<device_name> [<period> <device_func>] TSEMICOLON	[TIDENTIFIER, TKEYWORD, TDOT]
23.	<device_func>	:=	<pulse_func> [<coil_func> [<sensor_func> [<programmable_func> [<wait_time>	[KON [KOFF, KWAITON [KWAITOFF, KSTROBE, KDO [KSEND, TINTEGER]
24.	<procedure_name>	:=	<identifier>	[TIDENTIFIER]
24.	<program_stmtlist>	:=	<program_stmt> [<program_stmtlist>	[TIDENTIFIER, TKEYWORD]
25.	<program_stmt>	:=	<procedure_name> <cell_name> [(<repetition_clause>)]	[TIDENTIFIER, TLPAREN]
26.	<repetition_clause>	:=	<iterations> [<condition>	[TINTEGER, TIDENTIFIER]
27.	<iterations>	:=	<integer>	[TINTEGER]
28.	<condition>	:=	<cell_name> <period> <signal>	[TIDENTIFIER, TKEYWORD, TDOT]
29.	<signal>	:=	ON [OFF	
31.	<pulse_func>	:=	<period> KSTROBE	[TDOT, KSTROBE]
32.	<coil_func>	:=	<period> KON [KOFF	[TDOT, KON [KOFF]
33.	<sensor_func>	:=	<period> KWAITON [KWAITOFF	[TDOT, KWAITON [KWAITOFF]
34.	<programmable_func>	:=	<period> KSEND [KDO ({ <parameter>, . . . })	[TDOT, KSEND [KDO, TLPAREN]
35.	<parameter>	:=	<string> [<identifier>	[TSTRING, TIDENTIFIER]
36.	<wait_time>	:=	<period> <integer>	[TINTEGER]
37.	<nonpgble_type>	=	KCOIL [KSENSOR [KPULSE [KWAIT	
38.	<predefined_port>	=	KCTL [KCOM1	
39.	<valid_bit>	=	[2 [4 [8 [16 [32	
40.	<iterations>	=	<integer>	[TINTEGER]
41.	<integer>	=	TINTEGER	
42.	<identifier>	=	TIDENTIFIER	
43.	<string>	=	TSTRING	
44.	<open_parenthesis>	=	TLPAREN	
45.	<close_parenthesis>	=	TRPAREN	
46.	<period>	:=	TDOT	

APPENDIX E: Example CPL Source Program

```
* Port declarations
Ports
    PortC 64259 Output:
    PortA 64256 Input:
    Com1port COM1: 300 7 2 0:          * Serial port declaration
End

* Device declarations
Devices
    PalletLiftUp      Pulse          PortC    4:
    Conveyor          Coil           PortC    5:
    PhotoCell         Sensor         PortA    7:
    PalletArrived     Sensor         PortA    6:
    ChuckOpen         Pulse          PortC    1:
    LatheG66inp       Pulse          PortC    0:
    Robot             Programmable   LPT1:
    Lathe             Programmable   Com1port:
    LatheStart        Pulse          PortC    2:
    LatheStop         Sensor         PortA    4:
    PalletLifted      Sensor         PortA    5:
    PalletStops       Coil           PortC    0:
    ChuckClose        Pulse          PortC    3:
    PalletLiftDown    Pulse          PortC    6:
    LatheRunning      Sensor         PortA    2:
    LatheHandShk      Pulse          PortC    3:
    Delay             Wait:
End
```

```
* Procedure operations
Procedure
    LatheHandShk.Strobe;
    LatheG66inp.Strobe;
    Lathe.Do(loadlathe);
    Robot.Send("NT");
    PalletStops.On;
    Conveyor.On;
    PhotoCell.WaitOn;
    PalletStops.Off;
    PalletArrived.WaitOn;
    Delay.1000;
    PalletLiftUp.Strobe;
    PalletLifted.WaitOn;
    Conveyor.Off;
    ChuckOpen.Strobe;
    Robot.Do(Loadpart);
    Delay.1000;
    ChuckClose.Strobe;
    Delay.2000;
    Robot.Do(Moveaway);
    Delay.2000;
    LatheStart.Strobe;
    LatheStop.WaitOff;
    Robot.Do(Moveback);
    Delay.2000;
    ChuckOpen.Strobe;
    Delay.2000;
    Robot.Do(Getpart);
```

```
    PalletStops.On;
    PalletLiftDown.Strobe;
    Conveyor.On;
    Delay.500;
    Conveyor.Off;
    PalletStops.Off;
    LatheStart.Strobe;
    LatheHandShk.Strobe;
End;
```

APPENDIX F: CPL Compiler Source Code

```
//------------------------------[ main.cpp ]---------------------------------
/*  ************************************************************************ *
/
/*  This is the main program of the CPL compiler.  This program is copyrighte
d
    and belongs to the Applied Science Division of Miami University.  Copying
    source code is illegal.
*/
/* NAME:         Main
         AUTHOR:  Meghamala Nugehally
         DATE:    10/16/92
         VERSION:        4.0
*/
/*  ************************************************************************ */

#include <stdio.h>
#include "..\head\errors.h"
#include "..\head\trans.h"
#include "..\head\table.h"
#include "..\head\list.h"
#include "..\head\crossref.h"

table symbol_table(TBLSZ);                        // symbol table of known size
table* sytb = &symbol_table;
list_t* procPtr;                 // list of executable statements
error_t errhandler("..\\prj\\errors.dat");
error_t* ee = &errhandler;

int numports = 0;          // to be eliminated as global
int numtypes = 0;          // to be eliminated as global

/*  ************************************************************************ *
/
/* FUNCTION:    The main program instantiates the program as a translate
   object and calls the parsing functions to begin the compilation process.
   After the completion of parsing, crossreferences are built.
*/
/*  ************************************************************************ */

main(int argc, char* argv[])
{
    // if file name not entered, prompt user and get filename
         if (argc != 2)
                printf("Enter program file name: %s\n", gets(argv[1]));

                translate_t program(argv[1]);

      program.parsePorts();          // parse ports section
                program.parseDevices();        // parse device section
                program.parseProcedure();      // parse procedure section
                program.generate();            // Generate code
                program.crossrefer();          // Generate cross reference lis
ting
                if (ee->Errors() == 0)
                {
        printf("\n Compilation successfully completed\n");

                }
                else
                {
```

```
                        printf("Compilation completed with %d errors\n", ee->
Errors());
                }

}

/* end of file */
```

```
//-------------------------------[ char.h ]---------------------------------
~

#ifndef CHAR_H
#define CHAR_H

#define LINELEN 80

#include <stdio.h>

// The character class identifies different categories of characters in the
// ascii set. It reads characters from the source file and returns them to
// the scanner a character at a time.

class char_t {
    int reget_flag;                    // to provide for one character look
ahead
    int last_char;          // the character read from the file
    int nextchar;
    FILE *fp;
         char* nextline, *thisline;
public:
    char_t(FILE *f) { reget_flag = 0; last_char = 0; fp = f;
                                  nextchar = 0;
                                  nextline = new char[LINELEN];
                                  thisline = new char[LINELEN];
                                  Readline(); }
    void reget(void) { reget_flag = 1; };
    int code(void) { return(last_char); };
    int next(void);
    void Readline();
    void reset_nextchar() { nextchar = 0; }
         short Class(void);
         char* Nextline() { return (nextline); }
         char* Thisline() { return (thisline); }
};

/* Character Classes */

#define CILL    0       /* illegal character */
#define CWHITE  1       /* white space */
#define CQUOTE  2
#define CID     3       /* ok for identifiers */
#define CLPAREN 4
#define CRPAREN 5
#define CCOMMA  6
#define CDOT    7
#define CDIG    8
#define CCOLON  9
#define CSEMI   10
#define CLF     11      /* line feed */
#define CEOF    12      /* end of file */
#define CCOMMNT 13              /* comment character */


#endif
```

```
//--------------------------[ char.cpp ]-----------------------------------

/* This file initializes a table of valid character constants, and defines
        the member functions for the char class
*/

#include <string.h>
#include "..\head\char.h"
#include "..\head\errors.h"

extern error_t * ee;       // error object declared in main.cpp

short ch_class[] = {
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,          /* Nu,...*/
 CILL,CWHITE,CLF,CWHITE,CWHITE,CWHITE,CILL,CILL,   /* bs,ht,lf,vt,ff,cr,so,si*
/
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,
 CILL,CILL,CILL,CILL,CILL,CILL,CILL,CILL,
 CWHITE,CILL,CQUOTE,CID,CID,CID,CILL,CILL,         /* sp,!,",#,$,%,&,'*/
 CLPAREN,CRPAREN,CILL,CILL,CCOMMA,CILL,CDOT,CILL,  /* (,),*,+,,,-,.,/ */
 CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,CDIG,          /* 0,1,2,3,4,5,6,7 */
 CDIG,CDIG,CCOLON,CSEMI,CILL,CILL,CILL,CILL,       /* 8,9,:,;,<,=,>,? */
 CILL,CID,CID,CID,CID,CID,CID,CID,                 /* @,A,B,C,D,E,F,G */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* H,I,J,K,L,M,N,O */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* P,Q,R,S,T,U,V,W */
 CID,CID,CID,CILL,CILL,CILL,CILL,CID,              /* X,Y,Z,[,\,],^,_ */
 CILL,CID,CID,CID,CID,CID,CID,CID,                 /* `,a,b,c,d,e,f,g */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* h,i,j,k,l,m,n,o */
 CID,CID,CID,CID,CID,CID,CID,CID,                  /* p,q,r,s,t,u,v,w */
 CID,CID,CID,CILL,CILL,CILL,CILL,CILL              /* x,y,z,{,|,},~,del */
};

/*  ********************************************************************** *
/
/*
        FUNCTION:       Reads one source code line from the cpl program file
                                        Comments are ignored, and single char
acter backtracking is
                implemented.
*/
/*  ********************************************************************** *
/

void char_t::Readline()
{
        strcpy(thisline, nextline);      // make a copy of the current line
        if (fgets(nextline, LINELEN+1, fp) == NULL)        // read next line
        {       printf("%-80s", ee->readerror(_F_EREADSRCFIL));
                ee->checkerrors(FATAL_T);
        }

}
/*  ********************************************************************** */
/*
   FUNCTION:    If the reget flag is set, returns the previous character read
   else, reads the next character from the source file and returns it.
*/

/*  ********************************************************************** */
int char_t::next(void)
```

```
{
        if (reget_flag)
        {
                reget_flag = 0;     // Reset reget flag if set
        } else
        {
                if (nextline[nextchar] == '*')      // check for comment chara
cter
      {
                        last_char = '\n';
                        nextchar = 0;
                }
                else
                {       last_char = nextline[nextchar];
                        nextchar++;
      }
        }
        return(last_char);     // Return previous character if reget flag set
}
/*  ********************************************************************** *
/
/*
        FUNCTION:    Returns a unique character code that identifies the char
acter
*/
/*  ********************************************************************** *
/

// If the character read is not end-of-file, then return the character type,
// else return the end-of-file code.

short char_t::Class(void) { return((last_char != EOF) ?
                                                                        ch_c
lass[last_char] : CEOF); }

/*  ********************************************************************** *
/

/* end of file */
```

```
//-----------------------------[crossref.h]---------------------------------

#ifndef CROSSREF_H
#define CROSSREF_H

#include <stdio.h>

#define MAX_ID          30
#define MAXLEN          10
#define MAXSTR          80
#define SPACE_0 ""
#define SPACE_19        "                   "


typedef int ndxarr[10];

// device cross reference record
struct devarry
{
        char deviceName[MAX_ID+1];
        char devType[MAX_ID+1];
        int     bit;
        char outport[MAX_ID+1];
};

// port cross reference record
struct declarry
{
        char deviceName[MAX_ID+1];
        char portName[MAX_ID+1];
        int bit;
   char outport[MAX_ID+1];
};

// port cross reference list
struct crf_1
{
        char portname[MAX_ID+1];
        devarry devinfo[10];         // devices associated with a port variab
le
};

// device cross reference list
struct crf_2
{
        char typename[MAX_ID+1];
        declarry devtp[10];      // device variables associated with a device
 type
};

//cross reference class to build the cross reference table and print it
class crossref_t
{
        crf_1 ptod[10];
        crf_2 ttod[10];
        FILE *crp;

public:
        void build(ndxarr, ndxarr);
        void print(char*, ndxarr, ndxarr);
```

```
        void display(char* );
};
#endif
```

```
// --------------------------crossref.cpp---------------------------------
#include "..\head\crossref.h"
#include "..\head\table.h"
#include "..\head\errors.h"

extern table* sytb;
extern int numports;
extern int numtypes;
extern error_t *ee;


/*  ********************************************************************** *
/

/*
        FUNCTION:    Builds a cross reference between port variables and devi
ce
        variables.  For each port variable, a list of the devices that use th
e port
        including the device name, device types and bit number is created. Pr
ovides
        useful debugging information.

*/

/* ********************************************************************** *
/

void crossref_t::build(ndxarr ind1, ndxarr ind2)
{
        symbol_t* ss;
        int i = 0;              // index of the latest  port name added to pt
od
        int z=0;        // index of the latest type name added to ttod
        int j = 0;      // index of existing port name if matches current port
 name
        int  k, y;      // k = number of unique ports, y = number of unique ty
pes
        int x = 0;      // index of existing type name if matches current type
 name
        unsigned m=0;  // counter for # of devices for which cross reference
is built
   unsigned l=0;  // index for symbol table
        char portt[MAX_ID+1]; // temporary variable to hold  port name
        char typet[MAX_ID+1]; // temporary variable to hold  type name

        // Build the port and device cross reference
        // Get devices from the symbol table to build the cross
        // reference list

        while (((ss = sytb->getsymbol(l)) != NULL) && m<sytb->get_numOfdev())
        {
                if (ss->type == KDEVICES &&
                        ((((device_t*) (ss->symbol))->get_devType()) != KWAIT
))
                {
                        strcpy(typet,((device_t*) (ss->symbol))->get_typename
());
                        switch (((device_t*)(ss->symbol))->get_devType())
                        {
```

```
                                case KCOIL:
                                        strcpy(portt,((coil_t*)(((device_t*)
(ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KSENSOR:
                                        strcpy(portt,((sensor_t*)(((device_t*
) (ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KPULSE:
                                        strcpy(portt,((pulse_t*)(((device_t*)
 (ss->symbol))->getDevptr()))->get_portName());
                                        break;
                                case KPROGRAMMABLE:
                                        strcpy(portt,((programmable_t*)(((dev
ice_t*) (ss->symbol))->getDevptr()))->get_comportName());
                                        break;
                                default:
                                        break;
                        }
                        j=0;x=0;

                    // check if port variable already entered in list
                        while ((strcmp(portt,ptod[j].portname) != 0) && j<i)
j++;
          // check if type name already entered in list
                        while ((strcmp(typet,ttod[x].typename) != 0) && x<z)
x++;
                        // if port variable already in list, add device to
                        // corresponding port row
                        if (j<i)
                        {
                                k=ind1[j]+1;
                        }
                        // else add the port variable and the corresponding
                        // device variable to a new row.
                        else
                        {
                                strcpy(ptod[i].portname,portt);
                                k=0;i++;
                        }

          // do the same for the device cross reference list
                        if (x<z)
                                y = ind2[x]+1;
                        else
                        {
                                strcpy(ttod[z].typename,typet);
                                y=0;z++;
                        }
          // copy device name
                        strcpy(ptod[j].devinfo[k].deviceName,
                                        ((device_t*)(ss->symbol))->get_devic
eName());
                        strcpy(ptod[j].devinfo[k].devType,
                                        ((device_t*)(ss->symbol))->get_typen
ame());
                        strcpy(ttod[x].devtp[y].deviceName,
                                        ((device_t*)(ss->symbol))->get_devic
eName());
                        strcpy(ttod[x].devtp[y].portName,portt);
```

```
                        ind1[j] = k;  // # devices that use jth  port variabl
e is k
                        ind2[x] = y;  // # devices that are of xth type is y

                        // copy remaining information about devices
                        switch (((device_t*)(ss->symbol))->get_devType())
                        {
                                case KPROGRAMMABLE:
                                        strcpy(ttod[x].devtp[y].outport,
                                                ((programmable_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_comportName());
                                        strcpy(ptod[j].devinfo[k].outport,
                                                ((programmable_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_comportName());
                                        break;
                                case KCOIL:
                                        ptod[j].devinfo[k].bit = ((coil_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((coil_t*)(((d
evice_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                case KSENSOR:
                                        ptod[j].devinfo[k].bit = ((sensor_t*)
(((device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((sensor_t*)((
(device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                case KPULSE:
                                        ptod[j].devinfo[k].bit = ((pulse_t*)(
((device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        ttod[x].devtp[y].bit = ((pulse_t*)(((
device_t*)(ss->symbol))->getDevptr()))->get_bit();
                                        break;
                                default:
                                        break;
                        }
                        m++;l++;
#ifdef VERBOSE
                        printf("%s %s %s %d\n", ptod[j].portname,
                                        ptod[j].devinfo[k].deviceName,
                                        ptod[j].devinfo[k].devType,ptod[j].d
evinfo[k].bit);
#endif
                }
                else
                        l++;
        }
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Outputs the cross reference when the compilation is compl
ete
                                        to a seperate file that has the sourc
e file name but with an
                                        extentsion of .ref.
```

```
*/

/* ********************************************************************** *
/

void crossref_t::print(char* crfile, ndxarr p, ndxarr q)
{
        char space[20];       // number of spaces

        // create cross reference file with same filename as input file but w
ith
        // an extension of 'ref'
        if ((crp = fopen(crfile, "w")) == NULL)
        {       printf("%s\n", ee->readerror(_W_UNOPNCRFILE));
                ee->checkerrors(WARNING_T);
                // Error opening cross reference file\n");
   }

        else
        {
                fprintf(crp,"***********Cross references for ports and devic
es*********\n\n");

                for (int i=0; i <= numports; i++)
                {
                        fprintf(crp,"%-19s", ptod[i].portname);
                        strcpy(space, SPACE_0);       // contains a null stri
ng for the first line
                        for (int j=0; j<=p[i]; j++)
                        {
                                if (strcmp(ptod[i].devinfo[j].devType, "PROGR
AMMABLE") == 0)
                                        fprintf(crp,"%s %-14s %-15s\n", space
, ptod[i].devinfo[j].deviceName,
                                                ptod[i].devinfo[j].devType);
                                else
                                        fprintf(crp,"%s %-14s %-14s %d\n", sp
ace, ptod[i].devinfo[j].deviceName,
                                                ptod[i].devinfo[j].devType,
ptod[i].devinfo[j].bit);
                                strcpy(space, SPACE_19);  // contains 19 spac
es for all other lines

                        }
                }

                fprintf(crp,"\n***********Cross references for device types
and devices*********\n\n");

                for (int x=0; x<=numtypes; x++)
                {
                        fprintf(crp,"%-19s", ttod[x].typename);
                        strcpy(space, SPACE_0);    // contains  a null string
 for the first line

                        for (int y=0; y<=q[x]; y++)
                        {
                                if (strcmp(ttod[x].typename,"PROGRAMMABLE") =
= 0)
```

```
                                        fprintf(crp,"%s %-14s %-15s\n", space
, ttod[x].devtp[y].deviceName,
                                                ttod[x].devtp[y].portName);
                                else
                                        fprintf(crp,"%s %-14s %-14s %d\n", sp
ace, ttod[x].devtp[y].deviceName,
                                                ttod[x].devtp[y].portName, t
tod[x].devtp[y].bit);
                                strcpy(space, SPACE_19);    // contains 19 spa
ces for all other lines

                        }

                }
                fclose(crp);
        }
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Displays the cross reference listing on the screen at the
 end
                                        of compilation.

*/

/* ********************************************************************** *
/

void crossref_t::display(char* fn)
{
   char refstr[MAXSTR];

        if ((crp = fopen(fn, "r")) != NULL)
        {
                while (!feof(crp))
                {
                        if (fgets(refstr, MAXSTR, crp) != NULL)
                                printf("%s", refstr );
                }
                fclose(crp);
        }
        else {
                printf("%s\n", ee->readerror(_W_UNOPNCRFILE));
                ee->checkerrors(WARNING_T);
   }
}

/* end of file */
```

```
//---------------------------------errors.h-------------------------------

#ifndef ERRORS_H
#define ERRORS_H

#include <stdio.h>

#define codestring(errcode)      (#errcode)

#define MXLEN                              80
#define ERROR_T                 0
#define WARNING_T               1
#define FATAL_T                 2
#define _F_UNOPNSRCFIL  100
#define _F_UNOPNERRFIL  101
#define _F_UNOPNOUTFIL  102
#define _F_EREADSRCFIL 103
#define _F_PARSERERROR 104
#define _F_UNOPNCMDFIL 105
#define _F_UNREAERRFIL 106
#define _F_UNRESERRPTR 107
#define _E_SEMICOLEXPT  111
#define _E_FUNCOPREXPT  112
#define _E_UNDEFIDENTF 113
#define  _E_IDENTIFEXPT 114
#define  _E_TYPADDREXPT 115
#define _E_INCORSERPOR 116
#define _E_INTEGEREXPT 117
#define _E_INVALDEVTYP 118
#define _E_DEVTYPEXPTD 119
#define _E_INVALIDFUNC 120
#define _E_KEYWORDEXPT 121
#define _E_PARTYPMISMA 122
#define _E_PORTKEYEXPT  123
#define _E_DEVKEYWEXPT 124
#define _E_PROCKEYEXPT 125
#define _W_UNOPNCRFILE 150
#define _W_BAUDNOTSPEC  151
#define _W_STMTNOEFFEC 152
#define _W_UNOPNTRCFIL 153

class error_t {
        int err_type;           // FATAL/ERROR/WARNING
        int err_count;          // number of syntax errors
        int warn_count; // number of warning errors
        int tot_count;
        int err_limit;          // condition to terminate compilation
        int err_code;           // error code
        char* err_msg;          // error message
        FILE * ep;                      // error data file pointer
public:
        error_t(void);
        error_t(char* );
        char* readerror(int );
        void checkerrors(int);
        int Errcount() { return (err_count); }
        int Warncount() { return (warn_count); }
        int Errors() { return (tot_count); }
};
```

```
#endif
```

```
//------------------------------ errors.cpp ---------------------------------

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "..\head\errors.h"

/* ************************************************************************ *
/
/*
                FUNCTION:       The error class constructor initializes the e
rror object and
                allocates storage for it.  It opens the errors file errors.da
t.
/*
/* ************************************************************************ *
/

error_t::error_t(char* fname)
{
        if ((ep = fopen(fname, "r")) == NULL)
   {
                printf(" Unable to locate errors file "errors.dat\n");
                printf ("Copy "errors.dat" into the "prj" directory and recom
pile\n");
        }
        else
   {
                err_count = 0;
                warn_count = 0;
                tot_count = 0;
                err_limit = 5;
        }
}

/* ************************************************************************ *
/
/*
                FUNCTION:       This function retrieves error messages from t
he error file.
                The error message is matched with an error code which is pass
ed to the
      function by the calling routine.
/*
/* ************************************************************************ *
/

char* error_t::readerror(int ec) {

        char errstr[MXLEN];
        char revstr[MXLEN];
        char * msgstr, *codestr;
        char* p;

        if (fseek(ep, 0L, SEEK_SET) == 0)
        {
                while (!feof(ep))
                {       if ((fgets(errstr, MXLEN, ep)) != NULL)
                        {    sscanf(errstr, "%d", &err_code);
```

```
                                if (err_code == ec)
                                {       strrev(errstr);
                                        p = strrchr(errstr, ' ');
                                        *p = '\0';
                                        strrev(errstr);
                                        return (errstr);

                                }
                        }
                        else
                        {
                                printf("%s\n", readerror(_F_UNREAERRFIL));
                                checkerrors(FATAL_T);
         }
                }  // end of while
        }
        else
        {
                printf("%s\n", readerror(_F_UNRESERRPTR));
                checkerrors(FATAL_T);
        }
        return (NULL);
}

/* ********************************************************************** *
/
/*
                FUNCTION:       This function retrieves error messages from t
he error file.
                The error message is matched with an error code which is pass
ed to the
      function by the calling routine.
/*
/* ********************************************************************** *
/

void error_t::checkerrors(int ty)
{
        switch (ty)
        {
                case ERROR_T:
                        err_count++;
                        tot_count = err_count + warn_count;
                        if (tot_count > err_limit)
                        {       printf ("Compilation ended with %d errors\n",
 tot_count);
                                exit(-1);
                        }
                        break;
                case WARNING_T:
                        warn_count++;
                        tot_count = err_count + warn_count;
                        if (tot_count > err_limit)
                        {       printf ("Compilation ended with %d errors\n",
 tot_count);
                                exit(-1);
                        }
                        break;
                case FATAL_T:
                        printf("Fatal error discovered\n");
```

```
                        printf("Compilation cannot continue\n");
                        printf("%d errors discovered during compilation\n", t
ot_count);
                        exit(-2);
                        break;
                default:
                        break;
        }
}
```

```
// --------------------------FUNCTBL.H--------------------------------

#ifndef FUNC_H
#define FUNC_H

#define DEVINDEX                5
#define FUNCINDEX               10
#define MAXDEV                  5
#define MAXFUNC         8
#define MAXPARAM                10
#define CVALID                  1
#define CINVALID                0
#define NOPAR                   0
#define NA                              0

/* Function opcode definitions */

#define CON                             1
#define COFF                    2
#define CWAITON         3
#define CWAITOFF                4
#define CSEND                   5
#define CWAIT                   6
#define CSTROBE         7
#define CDO                             8
#define CSERPORT                9
#define CPORT                   10
#define CSOURCE         11

// define the opcode and parameters for a function and indicate if valid
// for a given device type
typedef struct {
        int valid;
        int opcode;
        int plist[MAXPARAM];     // list of parameter types
}funcInfo;

// define all functions for all device types
typedef funcInfo tbltype[MAXDEV][MAXFUNC];

#endif
```

```
// -------------------------[functbl.cpp]--------------------------------

#include "..\head\functbl.h"
#include "..\head\token.h"

/* Initialize function table with valid functions and parameters */
tbltype functbl =

/* coil device */             {  {
/* func ON     */                        { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func off    */                        { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func waiton */                        { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func waitoff*/                        { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func send    */                       { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func wait    */                       { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func strobe   */                      { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
/* func do      */                       { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                 },
/* sensor device */              {
                                         { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                 },
/* pulse device  */              {
                                         { CVALID, CON, {NOPAR, NOPAR, NOPAR,
NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, COFF, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CWAITON, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CWAITOFF, {NOPAR, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CSTROBE, {NOPAR, NOPAR, NOP
AR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
```

```
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                   },
/* for programmable */             {
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CSEND, {TCHARSTRING, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CDO, {TIDENTIFIER, NOPAR, N
OPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                   },
/* Plain functions */              {
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CVALID, CWAIT, {TINTEGER, NOPAR, NO
PAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } },
                                         { CINVALID, NA, {NOPAR, NOPAR, NOPAR,
 NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR, NOPAR } }
                                   }
                         };
/* END OF FILE */
```

```
//--------------------------------[ list.h ]----------------------------------
-
/* This is a generic list class -- change the type names and reuse.  */

#ifndef LIST_H
#define LIST_H

#include "..\head\stmts.h"

//  Change this typedef to make the list operate on other types
typedef procedure_t * objptr_t;

struct entry_t  {                  // doubly linked list entry
    objptr_t obj;
    struct entry_t *prev, *next;
    };

//  class definition for a list of "statement" objects.
class list_t
{

    public:
        list_t(void);
        ~list_t(void);

        void insert(objptr_t obj);
        void append(objptr_t obj);
        void remove(objptr_t obj);
        int  length(void);

        objptr_t first(void);
        objptr_t next(void);
        objptr_t last(void);
        objptr_t prev(void);

   private:
        entry_t head, tail, *cursor;
        int n_entries;

};


#endif

// end of file
```

```
//--------------------------------[ list.cpp ]--------------------------------
-

/* AUTHOR:           Charles Ames
        MAINTAINED BY:          Meghamala Nugehally
        DATE:                   02/20/92

        Member functions for the list class.  This list class has the
        property that elements are always added at the head.

*/


#include <stdio.h>
#include "..\head\list.h"

/*  ********************************************************************** *
/

/*
   FUNCTION:    This is the constructor function for the list class. It creat
es
   an empty list and initializes the head and tail pointers to null.
*/

/*  ********************************************************************** *
/
list_t::list_t(void)
{

        head.obj = ( objptr_t )NULL;
        head.next = &tail;
        head.prev = (entry_t *)NULL;

        tail.obj = ( objptr_t )NULL;
        tail.prev = &head;
        tail.next = (entry_t *)NULL;

        cursor = &head;
        n_entries = 0;

        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    This is the destructor function for the list class. It
   deallocates memory for the current list object.
*/
/*  ********************************************************************** *
/

list_t::~list_t(void)
{
        entry_t *temp;
        cursor = head.next;

        while (cursor != &tail)
        {
```

```
                temp = cursor;
                cursor = cursor->next;
                delete temp;
        }
        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    It inserts a list element at the head of the list and updates
   the length of the list.
*/
/*  ********************************************************************** *
/

void list_t::insert( objptr_t obj)
{
        cursor = head.next;
        head.next = new entry_t;
        head.next->obj = obj;
        head.next->next = cursor;
        head.next->prev = &head;
        cursor->prev = head.next;

        n_entries++;

        return;
}


/*  ********************************************************************** *
/

/*
   FUNCTION:    It inserts a list element at the tail of the list and updates
   the length of the list.
*/
/* ********************************************************************** *
/

void list_t::append(objptr_t obj)
{

        cursor = tail.prev;
        tail.prev = new entry_t;
        tail.prev->obj = obj;
        tail.prev->prev = cursor;
        tail.prev->next = &tail;
        cursor->next = tail.prev;

        n_entries++;

        return;
}

/*  ********************************************************************** *
/

/*
```

```
   FUNCTION:    It deletes the list element that matches the element passed t
o
   it as input parameter, and updates the length of the list.
*/
/* ********************************************************************** *
/

void list_t::remove( objptr_t obj)
{
        int deleted = 0;

        cursor = head.next;

        while (cursor != &tail && !deleted)
        {
                if (cursor->obj == obj)
                {
                        cursor->prev->next = cursor->next;
                        cursor->next->prev = cursor->prev;
                        cursor->obj = ( objptr_t )NULL;
                        delete cursor;
                        deleted = 1;
                        n_entries--;
                }
                else
                {
                        cursor = cursor->next;
                }

        }

        return;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element at the head of the list.
*/
/* ********************************************************************** *
/

objptr_t list_t::first(void)
{

    cursor = head.next;
         return cursor->obj;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element next to the current.
*/
/* ********************************************************************** *
/

objptr_t list_t::next(void)
{
         if (cursor != &tail) cursor = cursor->next;
```

```
        return cursor->obj;
}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the element at the tail of the list.
*/
/* ********************************************************************** *
/

objptr_t list_t::last(void)
{
        cursor = tail.prev;
        return cursor->obj;

}

/*  ********************************************************************** *
/
/*
       FUNCTION:    Returns the element before the current.
*/
/* ********************************************************************** *
/

objptr_t list_t::prev(void)
{
        if (cursor != &head) cursor = cursor->prev;
        return cursor->obj;

}

/*  ********************************************************************** *
/
/*
   FUNCTION:    Returns the length of the list.
*/
/* ********************************************************************** *
/

int list_t::length(void)
{
        return n_entries;

}
// end of file
```

```
//--------------------------------[ stmts.h ]---------------------------------
-

#ifndef STMTS_H
#define STMTS_H

#include <string.h>
#include "..\head\token.h"

#define EXTLEN  4

// Structure to hold the p-code for every executable procedure statement
struct exec_code {
        int opcode;           // function opcode
        long int address;     // port address
        int direction;        // input or output
        int bit;              // bit number in the port
        int bitValue;         // 1 or 0
        char* comport;        // serial or parallel (LPT1: or COM1:)
        int baudrate;         // used  now for the serial ports
        int startbit;         // used now for serial ports
        int stopbit;          // used for serial port
        int paritybit;                  // for serial port
        char* filename;       // command file name for programmable devices
        char* string;         // string to be output to programmable devices
        int waitsecs;         // delay time in milliseconds
};

//Abstract base class for statements, used in list_t
class stmt_t {
    char name[10];              // statement type
public:
    stmt_t(void) { ; }
    stmt_t (char *n) { strcpy(name,n); }
    char *nameOf(void) { return(name); }

        // functions may be redefined in inheriting classes, port_t, device_
t
        // and procedure_t
        virtual void parse(void) { ; }
        virtual void generate(void) { ; }
};

// port statement type
class port_t : public stmt_t {
    char portName[MAX_ID+1];
    unsigned int portAddress;
        int direction;          // use keyword types
        int type;             // ordinary or serial port
        char typename[MAX_ID+1];        // stores serial port name (COM1:, CO
M2:)
        int baudrate;         // baudrate if serial port
    // start, stop, & parity bits if serial port
        int startbit, stopbit, paritybit;

public:
    port_t(void) { ; }
    port_t(token_t *);
    char* get_portName() { return(portName); }
    unsigned int get_portAddress() { return (portAddress); }
```

```
        int get_direction() { return(direction); }
        int Baudrate() { return (baudrate); }
        int StartBit() { return (startbit); }
        int StopBit() { return (stopbit); }
        int ParityBit() { return (paritybit); }
        int get_type() { return (type); }
    char* Typename() { return (typename); }
};

// device statement type
class device_t : public stmt_t {
    char deviceName[MAX_ID+1];
        int deviceType;             // use keyword types
    char typename[MAX_ID+1];
    device_t* devicePtr;
public:
    device_t(void)  { ; }
    device_t(char* d) { strcpy(deviceName, d); }
    device_t(token_t *);
    char* get_deviceName() { return (deviceName); }
        int get_devType() { return (deviceType); }
        char* get_typename() { return(typename); }
    device_t* getDevptr() { return (devicePtr); }
        virtual void parse(void) { ; }          // defined in inheriting cla
sses
        virtual void generate(void) { ; }
};

// coil device type derived from device_t
class coil_t : public device_t
{
    char portName[MAX_ID+1];
        int bit;
public:
    coil_t(void);
    coil_t(token_t *);
    char* get_portName() { return(portName); }
        void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }     // to be defined
    int get_bit() { return (bit); }
};

// sensor device type derived from device_t
class sensor_t : public device_t
{
    char portName[MAX_ID+1];
        int bit;
public:
    sensor_t(void);
    sensor_t(token_t *);
    char* get_portName() { return(portName); }
    void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }                     // to be defined
        int get_bit() { return (bit); }
};

// pulse device type derived from device_t
class pulse_t : public device_t
{
    char portName[MAX_ID+1];
```

```
        int bit;
public:
    pulse_t(void);
    pulse_t(token_t *);
    char* get_portName() { return(portName); }
    void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }              // to be defined
        int get_bit() { return (bit); }
};

// programmable device type derived from device_t
class programmable_t : public device_t
{
        char* comportName;
        int porttype;
public:
    programmable_t(void);
    programmable_t(token_t* );
        void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }                    // to be defined
        char* get_comportName() { return (comportName); }
    int PortType() { return (porttype); }
//    int get_baudrate() { return (baudrate); }  // not needed now
};

// wait device type derived from device_t
class wait_t : public device_t {
        unsigned int millisecs;
public:
        wait_t(void);
        wait_t(token_t* );
     unsigned int get_secs() { return (millisecs); }
        void parse(token_t*, exec_code*);
        void generate(exec_code*) { ; }                // to be defined
};

// procedure statement type derived from stmt_t
class procedure_t : public stmt_t {
    device_t* devptr;
public:
    exec_code codes;            // use keyword types
    procedure_t(void);
    procedure_t(token_t *);
};

#endif
```

```
//------------------------------[ stmts.cpp ]-------------------------------
-

#include <string.h>
#include "..\head\stmts.h"
#include "..\head\token.h"
#include "..\head\table.h"
#include "..\head\functbl.h"
#include "..\head\trans.h"
#include "..\head\list.h"
#include "..\head\errors.h"

extern table*   sytb;          // symbol table
extern list_t* procPtr;       // executable statement list
extern tbltype functbl;       // table of valid functions and corresponding

// valid parameters.
extern error_t* ee;                              // error handler object


extern int numports, numtypes;
int coiltypes, sensortypes, pulsetypes, progtypes, waittypes; // to be elimin

/*  ****Methods for translating port, device and procedure declarations**** *
/

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a port object and parses a port declaration stat
ement
        according to the syntax defined in the CPL grammar.  If any error is
        discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

port_t::port_t(token_t *t) : stmt_t ("Port_t")
{
        int ii;

        if (t->Type() == TIDENTIFIER)                    /* check for port var
iable */
        {
                strcpy(portName,t->Identifier());
                if (t->Next() == TINTEGER)               /* check for port add
ress */
                {    type = ADDRPORT;
               portAddress = t->Integer();
                        if ((((ii = t->Next()) == TKEYWORD) &&      /* directi
on check */
                                  ((t->Keytype() == KINPUT) ||
                                   (t->Keytype() == KOUTPUT))))
                        {
                                direction = t->Keytype();
                                if (t->Next() != TSEMICOLON)        /* end of
statement */
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
```

```
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }
                        }
                        else  /* end of statement with default direction */
                                if (ii == TSEMICOLON) direction = KINPUT;
                                {
                                        numports++;
#ifdef  VERBOSE
                                        printf("Port parsed successfully\n");
#endif
                                        return;
                                }
                }
                else  /* check if serial port */
                        if (t->Type() == TKEYWORD)
                        {
                                if ((t->Keytype() == KCOM1) || (t->Keytype()
== KCOM2))
                                {
                                        type = SERPORT;
                                        strcpy(typename, t->Identifier());
                                        if (t->Next() == TINTEGER)        //
baud rate
                                        {
                                                baudrate = t->Integer();
                                                if (t->Next() == TINTEGER)
// number of data bits for COM port
                                                {       startbit = t->Integer
();
                                                        if (t->Next() == TINT
EGER)  // number of stop bits for COM port
                                                        {       stopbit = t->
Integer();
                                                                if (t->Next()
 == TINTEGER)           // parity for COM port
                                                                {       parit
ybit = t->Integer();
                                                                        if (t
->Next() == TSEMICOLON)  // end of statement
                                                                        {
#ifdef  VERBOSE
printf("Device parsed successfully\n");
#endif
numports++;
return;
                                                                        }
                                                                        else
                                                                        {
printf("\n%s", t->Thisline());
printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_SEMICOLEXPT));
ee->checkerrors(ERROR_T);
                                                                        }
                                                                }
                                                                else
```

```
                                                                {       print
f("\n%s", t->Thisline());
                                                                        print
f("Line %d %-80s", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                                        ee->c
heckerrors(ERROR_T);
                                                                }

                                                        }
                                                        else
                                                        {       printf("\n%s"
, t->Thisline());
                                                                printf("Line
%d %-80s", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                                ee->checkerro
rs(ERROR_T);
                                                        }

                                                }
                                                else
                                                {       printf("\n%s", t->Thi
sline());
                                                        printf("Line %d %-80s
", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                        ee->checkerrors(ERROR
_T);
                                                }


                                        }
                                        else
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_W_BAUDNOTSPEC));
                                                ee->checkerrors(WARNING_T);
                                        }

                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_INCORSERPOR));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_TYPADDREXPT));
                                ee->checkerrors(ERROR_T);
                        }


        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
```

```
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a device object and parses a device declaration
        statement according to the syntax defined in the CPL grammar.  If any
 error
        is discovered, an appropriate error message is printed out.
*/
/* ********************************************************************** *
/

device_t::device_t(token_t *t) : stmt_t ("Device_t")
{
        if (t->Type() == TIDENTIFIER)                                   // de
vice variable
        {
                strcpy(deviceName,t->Identifier());
                if (t->Next() == TKEYWORD)
// look for device type
                {
                        deviceType = t->Keytype();
                        switch (deviceType)
                        {
                        case KCOIL:                     // invoke coil object
 to continue parsing
                                        devicePtr = new coil_t(t);
                                        strcpy(typename, "COIL");
                                        if (coiltypes == 0) numtypes++; // In
crement only if new type
                                        coiltypes++;
                                        break;
                        case KSENSOR:                   // create sensor devi
ce
                                        devicePtr = new sensor_t(t);
                                        strcpy(typename, "SENSOR");
                                        if (sensortypes == 0) numtypes++; //
Increment only if new type
                                        sensortypes++;
                                        break;
                        case KPULSE:                    // create pulse devic
e
                                        devicePtr = new pulse_t(t);
                                        strcpy(typename, "PULSE");
                                        if (pulsetypes == 0) numtypes++; // I
ncrement only if new type
                                        pulsetypes++;
                                        break;
                        case KPROGRAMMABLE:  // create programmable device
                                        devicePtr = new programmable_t(t);
                                        strcpy(typename, "PROGRAMMABLE");
                                        if (progtypes == 0) numtypes++; // In
crement only if new type
                                        progtypes++;
                                        break;
                        case KWAIT:         // create wait device
```

```
                                        strcpy(typename, "WAIT");
                                        devicePtr = new wait_t(t);
                                        break;
                        OTHERWISE:    // indicates misplaced keyword or incor
rect definition
                                                        // in funciton tabl
e
                                        printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_INVALDEVTYP));
                                        ee->checkerrors(ERROR_T);

                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_DEVTYPEXPTD));
                        ee->checkerrors(ERROR_T);
        }
          }
}

/*  ********************************************************************** *
/
/*
        FUNCTION: Creates a coil object and parses a coil device type declara
tion
        according to the syntax defined in the CPL grammar.  If any error is
        discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

coil_t::coil_t(token_t* t) : device_t ("Coil_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for port identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
                {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
        // check for bit number
         {
                                bit = t->Integer();
            if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
                                printf("Device parsed successfully\n");
#endif
                                return;
                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
```

```
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a sensor object and parses a sensor device type
        declaration according to the syntax     defined in the CPL grammar.
If any
        error is discovered, an appropriate     error message is printed out.

*/
/* ********************************************************************** *
/

sensor_t::sensor_t(token_t* t) : device_t ("Sensor_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
                {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
         {
                                bit = t->Integer();
            if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
                                        printf("Device parsed successfully\n"
);
#endif
                                        return;
                                }
```

```
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else printf("Syntax error - incorrect device declaration\n");
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a pulse object and parses a coil device type
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

pulse_t::pulse_t(token_t* t) : device_t ("Pulse_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
           {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
                        {
                                bit = t->Integer();
             if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
```

```
                                        printf("Device parsed successfully\n"
);
#endif
                                        return;
                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a programmable object and parses a coil device t
ype
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

programmable_t::programmable_t(token_t* t) : device_t ("Programmable_t")
{
        int tt;
        symbol_t* sy;
        port_t* pt;

   // check for serial or parallel port address
        if (t->Next() == TKEYWORD)
```

```
        {
                comportName = new char[strlen(t->Identifier())+1];
                strcpy(comportName, t->Identifier());
                porttype = PARPORT;
                if (t->Next() == TSEMICOLON)  // end of statement
                {
#ifdef  VERBOSE
                        printf("Device parsed successfully\n");
#endif
                        return;
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_SEMICOLEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else // Serial port
   {
                if (t->Type() == TIDENTIFIER)
                {       comportName = new char[strlen(t->Identifier())+1];
                        strcpy(comportName, t->Identifier());
                        porttype = SERPORT;
                        if (t->Next() == TSEMICOLON)  // end of statement
                        {
#ifdef  VERBOSE
                                printf("Device parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }

        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a wait object and parses a wait device type
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
```

```
/* ************************************************************************ *
/

wait_t::wait_t(token_t* t) : device_t ("Wait")
{
        if (t->Next() == TSEMICOLON)
        {
#ifdef  VERBOSE
                printf("Device parsed successfully\n");
#endif
                return;
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_SEMIC
OLEXPT));
                ee->checkerrors(ERROR_T);
        }

}


/*  ************************************************************************ *
/

/*
        FUNCTION:    Creates a procedure object and parses an executeable sta
tement
        until a semicolon is read according to the syntax defined in the CPL
        grammar.  If any error is discovered, an appropriate error message is
        printed out.

*/
/* ************************************************************************ *
/
procedure_t::procedure_t(token_t *t) : stmt_t ("Procedure_t")
{
        objptr_t code;
        symbol_t* st;
        device_t* dt;

        if (t->Type() == TIDENTIFIER)
        {
                // search for device name in symbol table
                if (!((st = sytb->search(NULL, t->Identifier(), KDEVICES, 0))
 == NULL))
                {
                        devptr = (device_t*) st->symbol;
                        // typecast to specify device type --can be eliminate
d
                        switch (devptr->get_devType())
                        {
                                case KCOIL:
                                        ((coil_t*) (devptr->getDevptr()))->pa
rse(t, devptr, &codes);
                                        break;
                                case KSENSOR:
                                        ((sensor_t*) (devptr->getDevptr()))->
parse(t, devptr, &codes);
                                        break;
```

```
                                case KPULSE:
                                        ((pulse_t*) (devptr->getDevptr()))->p
arse(t, devptr, &codes);
                                        break;
                                case KPROGRAMMABLE:
                                        ((programmable_t*) (devptr->getDevptr
()))->parse(t, devptr, &codes);
                                        break;
                                case KWAIT:
                                        ((wait_t*) (devptr->getDevptr()))->pa
rse(t, &codes);
                                        break;
                                default:
                                        break;
                        }
                }
        else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void coil_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton with valid parameters
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)  // shou
ld be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
```

```
                                                    // search through the functio
n table
                                                    if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                            (valid = functbl[i][j
].valid))
                                                    {
                                                            code->opcode = functb
l[i][j].opcode;
                                                            code->address = ((por
t_t*)((sytb->search(NULL,
                                                            ((coil_t* )(dt->getDe
vptr()))->get_portName(),
                                                            KPORTS,0))->symbol))-
>get_portAddress();
                                                            code->bit = bit;
                                                    }
                                            } // end of first for
                                    }
                            }  // end of second for
                            if (!valid)
                            {       printf("\n%s", t->Thisline());
                                    printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                    ee->checkerrors(ERROR_T);
                            }
                 if (t->Next() == TSEMICOLON)
                            {
#ifdef  VERBOSE
                                    printf("Statement parsed successfully\n");
#endif
                                    return;
                            }
                            else
                            {       printf("\n%s", t->Thisline());
                                    printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                    ee->checkerrors(ERROR_T);
                            }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                        ee->checkerrors(WARNING_T);
                }

}

/*  ********************************************************************** *
/
```

```
/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void sensor_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                                                {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        code->address = ((por
t_t*)((sytb->search(NULL,
                                                        ((sensor_t* )(dt->get
Devptr()))->get_portName(),
                                                        KPORTS,0))->symbol))-
>get_portAddress();
                                                        code->bit = bit;
                                                }
                                        }    // end of first for
                                }
                        }   // end of second for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }

                        if (t->Next() == TSEMICOLON)
          {
#ifdef  VERBOSE
                                printf("Statement parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
```

```
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void pulse_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                                                {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        code->address = ((por
t_t*)((sytb->search(NULL,
```

```
                                                        ((pulse_t* )(dt->getD
evptr()))->get_portName(),
                                                        KPORTS,0))->symbol))-
>get_portAddress();
                                                        code->bit = bit;
                                                }
                                        }    // end of first for
                                }
                        }        // end of second for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }
                        if (t->Next() == TSEMICOLON)
         {
#ifdef  VERBOSE
                                printf("Statement parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/*  ********************************************************************** *
/

/*
        FUNCTION:    Parses a programmable device type declaration according
to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/
```

```
void programmable_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int nn;
        int valid = 0;
        int validparam = 0;
        port_t* pp;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton with valid parameters
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                  {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        switch (porttype)
                                                        {
                                                                case SERPORT:
                                                                        pp =
sytb->convertPort((sytb->search(NULL,
        comportName, KPORTS, 0))->symbol);
                                                                        code-
>comport = new char[strlen(pp->Typename())+1];
                                                                        strcp
y(code->comport, pp->Typename());
                                                                        break
;
                                                                case PARPORT:
                                                                        code-
>comport = new char[strlen(comportName)+1];
                                                                        strcp
y(code->comport, comportName);
                                                                        break
;
                                                                default:
                                                                        break
;
                                                        }
                                                }
                                        }  // end of 2nd for
                                        if (!valid)
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_E_INVALIDFUNC));
                                                ee->checkerrors(ERROR_T);
                                        }
                                        // check for valid parameters
```

```
                                                if ((nn = t->Next()) == TLPAREN)
                                                {
                                                        int k = 0;
                                                        while ((t->Next() != TRPAREN)
 && ( !validparam || k < MAXPARAM))
                                                        {
                                                                if (t->Type() == TCOM
MA || t->Type() == TQUOTE)
                                                                        t->Next();
                                                                if (functbl[i][j-1].p
list[k] == t->Type())
                                                                {
                                                                        if (t->Type()
 == TCHARSTRING)
                                                                        {
                                                                                code-
>string = new char[strlen(t->Identifier())+1];
                                                                                strcp
y(code->string, t->Identifier());
                                                                        }
                                                                        if (t->Type()
 == TIDENTIFIER)
                                                                        {
                                                                                code-
>filename = new char[strlen(t->Identifier())+1];
                                                                                strcp
y(code->filename, t->Identifier());
                                                                        }
                                                                        validparam =
!validparam;
                                                                        k++;
                                                                }
                                                                else
                                                                {       validparam =
CINVALID;
                                                                        printf("\n%s"
, t->Thisline());
                                                                        printf("Line
%d %-80s", t->Line_num(), ee->readerror(_E_PARTYPMISMA));
                                                                        ee->checkerro
rs(ERROR_T);
                                                                }
                                                        } // END OF WHILE
                                                        if (t->Next() == TSEMICOLON)
                    {
#ifdef  VERBOSE
                                                                printf("Procedure sta
tement parsed successfully\n");
#endif
                                                                return;
                                                        }
                                                        else
                                                        {       printf("\n%s", t->Thi
sline());
                                                                printf("Line %d %-80s
", t->Line_num(), ee->readerror(_E_SEMICOLEXPT));
                                                                ee->checkerrors(ERROR
_T);
                                                        }
```

```
                                        }  // end of if
                                        else
                                        if (nn != TSEMICOLON)
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_E_SEMICOLEXPT));
                                                ee->checkerrors(ERROR_T);
                                        }

                                }  // end of IF checking for device index
                        } // end of 1st for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}


/*  ********************************************************************** *
/

/*
        FUNCTION:    Parses a wait device type declaration according to the s
yntax
        defined in the CPL grammar.  If any error is discovered, an appropria
te
   error message is printed out.

*/

/* ********************************************************************** *
/
void wait_t::parse(token_t* t, exec_code* code)
{
        code->opcode = CWAIT;
        if (t->Next() == TDOT)
        {
        // delay time in milliseconds
                if (t->Next() == TINTEGER)
                {
```

```
                        code->waitsecs = t->Integer();
                        millisecs = t->Integer();
                        if (t->Next() != TSEMICOLON)
                // end of statement
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_INTEGEREXPT));
                        ee->checkerrors(ERROR_T);
                }

   }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/* end of file */
```

```
//--------------------------------[ stmts.h ]---------------------------------
-

#ifndef STMTS_H
#define STMTS_H

#include <string.h>
#include "..\head\token.h"

#define EXTLEN  4

// Structure to hold the p-code for every executable procedure statement
struct exec_code {
        int opcode;           // function opcode
        long int address;     // port address
        int direction;        // input or output
        int bit;              // bit number in the port
        int bitValue;         // 1 or 0
        char* comport;        // serial or parallel (LPT1: or COM1:)
        int baudrate;         // used  now for the serial ports
        int startbit;         // used now for serial ports
        int stopbit;          // used for serial port
        int paritybit;                  // for serial port
        char* filename;       // command file name for programmable devices
        char* string;         // string to be output to programmable devices
        int waitsecs;         // delay time in milliseconds
};

//Abstract base class for statements, used in list_t
class stmt_t {
    char name[10];              // statement type
public:
    stmt_t(void) { ; }
    stmt_t (char *n) { strcpy(name,n); }
    char *nameOf(void) { return(name); }

        // functions may be redefined in inheriting classes, port_t, device_
t
        // and procedure_t
        virtual void parse(void) { ; }
        virtual void generate(void) { ; }
};

// port statement type
class port_t : public stmt_t {
    char portName[MAX_ID+1];
    unsigned int portAddress;
        int direction;          // use keyword types
        int type;             // ordinary or serial port
        char typename[MAX_ID+1];        // stores serial port name (COM1:, CO
M2:)
        int baudrate;         // baudrate if serial port
    // start, stop, & parity bits if serial port
        int startbit, stopbit, paritybit;

public:
    port_t(void) { ; }
    port_t(token_t *);
    char* get_portName() { return(portName); }
    unsigned int get_portAddress() { return (portAddress); }
```

```
        int get_direction() { return(direction); }
        int Baudrate() { return (baudrate); }
        int StartBit() { return (startbit); }
        int StopBit() { return (stopbit); }
        int ParityBit() { return (paritybit); }
        int get_type() { return (type); }
    char* Typename() { return (typename); }
};

// device statement type
class device_t : public stmt_t {
    char deviceName[MAX_ID+1];
        int deviceType;             // use keyword types
    char typename[MAX_ID+1];
    device_t* devicePtr;
public:
    device_t(void)  { ; }
    device_t(char* d) { strcpy(deviceName, d); }
    device_t(token_t *);
    char* get_deviceName() { return (deviceName); }
        int get_devType() { return (deviceType); }
        char* get_typename() { return(typename); }
    device_t* getDevptr() { return (devicePtr); }
        virtual void parse(void) { ; }          // defined in inheriting cla
sses
        virtual void generate(void) { ; }
};

// coil device type derived from device_t
class coil_t : public device_t
{
    char portName[MAX_ID+1];
        int bit;
public:
    coil_t(void);
    coil_t(token_t *);
    char* get_portName() { return(portName); }
        void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }     // to be defined
    int get_bit() { return (bit); }
};

// sensor device type derived from device_t
class sensor_t : public device_t
{
    char portName[MAX_ID+1];
        int bit;
public:
    sensor_t(void);
    sensor_t(token_t *);
    char* get_portName() { return(portName); }
    void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }                     // to be defined
        int get_bit() { return (bit); }
};

// pulse device type derived from device_t
class pulse_t : public device_t
{
    char portName[MAX_ID+1];
```

```
        int bit;
public:
    pulse_t(void);
    pulse_t(token_t *);
    char* get_portName() { return(portName); }
    void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }              // to be defined
        int get_bit() { return (bit); }
};

// programmable device type derived from device_t
class programmable_t : public device_t
{
        char* comportName;
        int porttype;
public:
    programmable_t(void);
    programmable_t(token_t* );
        void parse(token_t*, device_t*, exec_code*);
    void generate(exec_code*) { ; }                    // to be defined
        char* get_comportName() { return (comportName); }
    int PortType() { return (porttype); }
//    int get_baudrate() { return (baudrate); }  // not needed now
};

// wait device type derived from device_t
class wait_t : public device_t {
        unsigned int millisecs;
public:
        wait_t(void);
        wait_t(token_t* );
     unsigned int get_secs() { return (millisecs); }
        void parse(token_t*, exec_code*);
        void generate(exec_code*) { ; }                // to be defined
};

// procedure statement type derived from stmt_t
class procedure_t : public stmt_t {
    device_t* devptr;
public:
    exec_code codes;            // use keyword types
    procedure_t(void);
    procedure_t(token_t *);
};

#endif
```

```
//------------------------------[ stmts.cpp ]-------------------------------
-

#include <string.h>
#include "..\head\stmts.h"
#include "..\head\token.h"
#include "..\head\table.h"
#include "..\head\functbl.h"
#include "..\head\trans.h"
#include "..\head\list.h"
#include "..\head\errors.h"

extern table*   sytb;          // symbol table
extern list_t* procPtr;       // executable statement list
extern tbltype functbl;       // table of valid functions and corresponding

// valid parameters.
extern error_t* ee;                              // error handler object


extern int numports, numtypes;
int coiltypes, sensortypes, pulsetypes, progtypes, waittypes; // to be elimin

/*  ****Methods for translating port, device and procedure declarations**** *
/

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a port object and parses a port declaration stat
ement
        according to the syntax defined in the CPL grammar.  If any error is
        discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

port_t::port_t(token_t *t) : stmt_t ("Port_t")
{
        int ii;

        if (t->Type() == TIDENTIFIER)                    /* check for port var
iable */
        {
                strcpy(portName,t->Identifier());
                if (t->Next() == TINTEGER)               /* check for port add
ress */
                {    type = ADDRPORT;
               portAddress = t->Integer();
                        if ((((ii = t->Next()) == TKEYWORD) &&      /* directi
on check */
                                  ((t->Keytype() == KINPUT) ||
                                   (t->Keytype() == KOUTPUT))))
                        {
                                direction = t->Keytype();
                                if (t->Next() != TSEMICOLON)        /* end of
statement */
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
```

```
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }
                        }
                        else  /* end of statement with default direction */
                                if (ii == TSEMICOLON) direction = KINPUT;
                                {
                                        numports++;
#ifdef  VERBOSE
                                        printf("Port parsed successfully\n");
#endif
                                        return;
                                }
                }
                else  /* check if serial port */
                        if (t->Type() == TKEYWORD)
                        {
                                if ((t->Keytype() == KCOM1) || (t->Keytype()
== KCOM2))
                                {
                                        type = SERPORT;
                                        strcpy(typename, t->Identifier());
                                        if (t->Next() == TINTEGER)        //
baud rate
                                        {
                                                baudrate = t->Integer();
                                                if (t->Next() == TINTEGER)
// number of data bits for COM port
                                                {       startbit = t->Integer
();
                                                        if (t->Next() == TINT
EGER)  // number of stop bits for COM port
                                                        {       stopbit = t->
Integer();
                                                                if (t->Next()
 == TINTEGER)           // parity for COM port
                                                                {       parit
ybit = t->Integer();
                                                                        if (t
->Next() == TSEMICOLON)  // end of statement
                                                                        {
#ifdef  VERBOSE
printf("Device parsed successfully\n");
#endif
numports++;
return;
                                                                        }
                                                                        else
                                                                        {
printf("\n%s", t->Thisline());
printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_SEMICOLEXPT));
ee->checkerrors(ERROR_T);
                                                                        }
                                                                }
                                                                else
```

```
                                                                {       print
f("\n%s", t->Thisline());
                                                                        print
f("Line %d %-80s", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                                        ee->c
heckerrors(ERROR_T);
                                                                }

                                                        }
                                                        else
                                                        {       printf("\n%s"
, t->Thisline());
                                                                printf("Line
%d %-80s", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                                ee->checkerro
rs(ERROR_T);
                                                        }

                                                }
                                                else
                                                {       printf("\n%s", t->Thi
sline());
                                                        printf("Line %d %-80s
", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                        ee->checkerrors(ERROR
_T);
                                                }


                                        }
                                        else
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_W_BAUDNOTSPEC));
                                                ee->checkerrors(WARNING_T);
                                        }

                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_INCORSERPOR));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_TYPADDREXPT));
                                ee->checkerrors(ERROR_T);
                        }


        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
```

```
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a device object and parses a device declaration
        statement according to the syntax defined in the CPL grammar.  If any
 error
        is discovered, an appropriate error message is printed out.
*/
/* ********************************************************************** *
/

device_t::device_t(token_t *t) : stmt_t ("Device_t")
{
        if (t->Type() == TIDENTIFIER)                                   // de
vice variable
        {
                strcpy(deviceName,t->Identifier());
                if (t->Next() == TKEYWORD)
// look for device type
                {
                        deviceType = t->Keytype();
                        switch (deviceType)
                        {
                        case KCOIL:                     // invoke coil object
 to continue parsing
                                        devicePtr = new coil_t(t);
                                        strcpy(typename, "COIL");
                                        if (coiltypes == 0) numtypes++; // In
crement only if new type
                                        coiltypes++;
                                        break;
                        case KSENSOR:                   // create sensor devi
ce
                                        devicePtr = new sensor_t(t);
                                        strcpy(typename, "SENSOR");
                                        if (sensortypes == 0) numtypes++; //
Increment only if new type
                                        sensortypes++;
                                        break;
                        case KPULSE:                    // create pulse devic
e
                                        devicePtr = new pulse_t(t);
                                        strcpy(typename, "PULSE");
                                        if (pulsetypes == 0) numtypes++; // I
ncrement only if new type
                                        pulsetypes++;
                                        break;
                        case KPROGRAMMABLE:  // create programmable device
                                        devicePtr = new programmable_t(t);
                                        strcpy(typename, "PROGRAMMABLE");
                                        if (progtypes == 0) numtypes++; // In
crement only if new type
                                        progtypes++;
                                        break;
                        case KWAIT:         // create wait device
```

```
                                        strcpy(typename, "WAIT");
                                        devicePtr = new wait_t(t);
                                        break;
                        OTHERWISE:    // indicates misplaced keyword or incor
rect definition
                                                        // in funciton tabl
e
                                        printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_INVALDEVTYP));
                                        ee->checkerrors(ERROR_T);

                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_DEVTYPEXPTD));
                        ee->checkerrors(ERROR_T);
        }
          }
}

/*  ********************************************************************** *
/
/*
        FUNCTION: Creates a coil object and parses a coil device type declara
tion
        according to the syntax defined in the CPL grammar.  If any error is
        discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

coil_t::coil_t(token_t* t) : device_t ("Coil_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for port identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
                {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
        // check for bit number
         {
                                bit = t->Integer();
            if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
                                printf("Device parsed successfully\n");
#endif
                                return;
                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
```

```
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a sensor object and parses a sensor device type
        declaration according to the syntax     defined in the CPL grammar.
If any
        error is discovered, an appropriate     error message is printed out.

*/
/* ********************************************************************** *
/

sensor_t::sensor_t(token_t* t) : device_t ("Sensor_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
                {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
         {
                                bit = t->Integer();
            if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
                                        printf("Device parsed successfully\n"
);
#endif
                                        return;
                                }
```

```
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else printf("Syntax error - incorrect device declaration\n");
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a pulse object and parses a coil device type
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

pulse_t::pulse_t(token_t* t) : device_t ("Pulse_t")
{
        if (t->Next() == TIDENTIFIER)
        {
                // search for identifier in symbol table
                if (!(sytb->search(NULL, t->Identifier(), KPORTS, 0) == NULL)
)
           {
                        strcpy(portName, t->Identifier());
                        if (t->Next() == TINTEGER)
                        {
                                bit = t->Integer();
             if (t->Next() == TSEMICOLON)
                                {
#ifdef  VERBOSE
```

```
                                        printf("Device parsed successfully\n"
);
#endif
                                        return;
                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num()
, ee->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INTEGEREXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                // if not declared print error
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a programmable object and parses a coil device t
ype
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
/* ********************************************************************** *
/

programmable_t::programmable_t(token_t* t) : device_t ("Programmable_t")
{
        int tt;
        symbol_t* sy;
        port_t* pt;

   // check for serial or parallel port address
        if (t->Next() == TKEYWORD)
```

```
        {
                comportName = new char[strlen(t->Identifier())+1];
                strcpy(comportName, t->Identifier());
                porttype = PARPORT;
                if (t->Next() == TSEMICOLON)  // end of statement
                {
#ifdef  VERBOSE
                        printf("Device parsed successfully\n");
#endif
                        return;
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_SEMICOLEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else // Serial port
   {
                if (t->Type() == TIDENTIFIER)
                {       comportName = new char[strlen(t->Identifier())+1];
                        strcpy(comportName, t->Identifier());
                        porttype = SERPORT;
                        if (t->Next() == TSEMICOLON)  // end of statement
                        {
#ifdef  VERBOSE
                                printf("Device parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }

        }

}

/*  ********************************************************************** *
/
/*
        FUNCTION:    Creates a wait object and parses a wait device type
        declaration according to the syntax defined in the CPL grammar.  If a
ny
        error is discovered, an appropriate error message is printed out.

*/
```

```
/* ************************************************************************ *
/

wait_t::wait_t(token_t* t) : device_t ("Wait")
{
        if (t->Next() == TSEMICOLON)
        {
#ifdef  VERBOSE
                printf("Device parsed successfully\n");
#endif
                return;
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_SEMIC
OLEXPT));
                ee->checkerrors(ERROR_T);
        }

}


/*  ************************************************************************ *
/

/*
        FUNCTION:    Creates a procedure object and parses an executeable sta
tement
        until a semicolon is read according to the syntax defined in the CPL
        grammar.  If any error is discovered, an appropriate error message is
        printed out.

*/
/* ************************************************************************ *
/
procedure_t::procedure_t(token_t *t) : stmt_t ("Procedure_t")
{
        objptr_t code;
        symbol_t* st;
        device_t* dt;

        if (t->Type() == TIDENTIFIER)
        {
                // search for device name in symbol table
                if (!((st = sytb->search(NULL, t->Identifier(), KDEVICES, 0))
 == NULL))
                {
                        devptr = (device_t*) st->symbol;
                        // typecast to specify device type --can be eliminate
d
                        switch (devptr->get_devType())
                        {
                                case KCOIL:
                                        ((coil_t*) (devptr->getDevptr()))->pa
rse(t, devptr, &codes);
                                        break;
                                case KSENSOR:
                                        ((sensor_t*) (devptr->getDevptr()))->
parse(t, devptr, &codes);
                                        break;
```

```
                                case KPULSE:
                                        ((pulse_t*) (devptr->getDevptr()))->p
arse(t, devptr, &codes);
                                        break;
                                case KPROGRAMMABLE:
                                        ((programmable_t*) (devptr->getDevptr
()))->parse(t, devptr, &codes);
                                        break;
                                case KWAIT:
                                        ((wait_t*) (devptr->getDevptr()))->pa
rse(t, &codes);
                                        break;
                                default:
                                        break;
                        }
                }
        else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_UNDEFIDENTF));
                        ee->checkerrors(ERROR_T);
        }
        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_E_IDENT
IFEXPT));
                ee->checkerrors(ERROR_T);
        }

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void coil_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton with valid parameters
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)  // shou
ld be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
```

```
                                                    // search through the functio
n table
                                                    if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                            (valid = functbl[i][j
].valid))
                                                    {
                                                            code->opcode = functb
l[i][j].opcode;
                                                            code->address = ((por
t_t*)((sytb->search(NULL,
                                                            ((coil_t* )(dt->getDe
vptr()))->get_portName(),
                                                            KPORTS,0))->symbol))-
>get_portAddress();
                                                            code->bit = bit;
                                                    }
                                            } // end of first for
                                    }
                            }  // end of second for
                            if (!valid)
                            {       printf("\n%s", t->Thisline());
                                    printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                    ee->checkerrors(ERROR_T);
                            }
                 if (t->Next() == TSEMICOLON)
                            {
#ifdef  VERBOSE
                                    printf("Statement parsed successfully\n");
#endif
                                    return;
                            }
                            else
                            {       printf("\n%s", t->Thisline());
                                    printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                    ee->checkerrors(ERROR_T);
                            }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                        ee->checkerrors(WARNING_T);
                }

}

/*  ********************************************************************** *
/
```

```
/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void sensor_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                                                {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        code->address = ((por
t_t*)((sytb->search(NULL,
                                                        ((sensor_t* )(dt->get
Devptr()))->get_portName(),
                                                        KPORTS,0))->symbol))-
>get_portAddress();
                                                        code->bit = bit;
                                                }
                                        }    // end of first for
                                }
                        }   // end of second for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }

                        if (t->Next() == TSEMICOLON)
          {
#ifdef  VERBOSE
                                printf("Statement parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
```

```
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Parses a coil device type declaration according to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/

void pulse_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int valid = 0;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                                                {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        code->address = ((por
t_t*)((sytb->search(NULL,
```

```
                                                        ((pulse_t* )(dt->getD
evptr()))->get_portName(),
                                                        KPORTS,0))->symbol))-
>get_portAddress();
                                                        code->bit = bit;
                                                }
                                        }    // end of first for
                                }
                        }        // end of second for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }
                        if (t->Next() == TSEMICOLON)
         {
#ifdef  VERBOSE
                                printf("Statement parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/*  ********************************************************************** *
/

/*
        FUNCTION:    Parses a programmable device type declaration according
to the syntax
   defined in the CPL grammar.  If any error is discovered, an appropriate
   error message is printed out.

*/

/* ********************************************************************** *
/
```

```
void programmable_t::parse(token_t* t, device_t* dt, exec_code* code)
{
        int nn;
        int valid = 0;
        int validparam = 0;
        port_t* pp;

        if (t->Next() == TDOT)
        {
                if (t->Next() == TKEYWORD)
                {       // check if valid funciton with valid parameters
                        for(int i=0;!valid&&i<MAXDEV;i++)
                        {
                                if (dt->get_devType() == i+DEVINDEX)
                // should be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                if ((t->Keytype() == j+FUNCIN
DEX) &&
                                                        (valid = functbl[i][j
].valid))
                  {
                                                        code->opcode = functb
l[i][j].opcode;
                                                        switch (porttype)
                                                        {
                                                                case SERPORT:
                                                                        pp =
sytb->convertPort((sytb->search(NULL,
        comportName, KPORTS, 0))->symbol);
                                                                        code-
>comport = new char[strlen(pp->Typename())+1];
                                                                        strcp
y(code->comport, pp->Typename());
                                                                        break
;
                                                                case PARPORT:
                                                                        code-
>comport = new char[strlen(comportName)+1];
                                                                        strcp
y(code->comport, comportName);
                                                                        break
;
                                                                default:
                                                                        break
;
                                                        }
                                                }
                                        }  // end of 2nd for
                                        if (!valid)
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_E_INVALIDFUNC));
                                                ee->checkerrors(ERROR_T);
                                        }
                                        // check for valid parameters
```

```
                                                if ((nn = t->Next()) == TLPAREN)
                                                {
                                                        int k = 0;
                                                        while ((t->Next() != TRPAREN)
 && ( !validparam || k < MAXPARAM))
                                                        {
                                                                if (t->Type() == TCOM
MA || t->Type() == TQUOTE)
                                                                        t->Next();
                                                                if (functbl[i][j-1].p
list[k] == t->Type())
                                                                {
                                                                        if (t->Type()
 == TCHARSTRING)
                                                                        {
                                                                                code-
>string = new char[strlen(t->Identifier())+1];
                                                                                strcp
y(code->string, t->Identifier());
                                                                        }
                                                                        if (t->Type()
 == TIDENTIFIER)
                                                                        {
                                                                                code-
>filename = new char[strlen(t->Identifier())+1];
                                                                                strcp
y(code->filename, t->Identifier());
                                                                        }
                                                                        validparam =
!validparam;
                                                                        k++;
                                                                }
                                                                else
                                                                {       validparam =
CINVALID;
                                                                        printf("\n%s"
, t->Thisline());
                                                                        printf("Line
%d %-80s", t->Line_num(), ee->readerror(_E_PARTYPMISMA));
                                                                        ee->checkerro
rs(ERROR_T);
                                                                }
                                                        } // END OF WHILE
                                                        if (t->Next() == TSEMICOLON)
                    {
#ifdef  VERBOSE
                                                                printf("Procedure sta
tement parsed successfully\n");
#endif
                                                                return;
                                                        }
                                                        else
                                                        {       printf("\n%s", t->Thi
sline());
                                                                printf("Line %d %-80s
", t->Line_num(), ee->readerror(_E_SEMICOLEXPT));
                                                                ee->checkerrors(ERROR
_T);
                                                        }
```

```
                                        }  // end of if
                                        else
                                        if (nn != TSEMICOLON)
                                        {       printf("\n%s", t->Thisline())
;
                                                printf("Line %d %-80s", t->Li
ne_num(), ee->readerror(_E_SEMICOLEXPT));
                                                ee->checkerrors(ERROR_T);
                                        }

                                }  // end of IF checking for device index
                        } // end of 1st for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_KEYWORDEXPT));
                        ee->checkerrors(ERROR_T);
                }

        }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}


/*  ********************************************************************** *
/

/*
        FUNCTION:    Parses a wait device type declaration according to the s
yntax
        defined in the CPL grammar.  If any error is discovered, an appropria
te
   error message is printed out.

*/

/* ********************************************************************** *
/
void wait_t::parse(token_t* t, exec_code* code)
{
        code->opcode = CWAIT;
        if (t->Next() == TDOT)
        {
        // delay time in milliseconds
                if (t->Next() == TINTEGER)
                {
```

```
                        code->waitsecs = t->Integer();
                        millisecs = t->Integer();
                        if (t->Next() != TSEMICOLON)
                // end of statement
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->re
aderror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }
                }
                else
                {       printf("\n%s", t->Thisline());
                        printf("Line %d %-80s", t->Line_num(), ee->readerror(
_E_INTEGEREXPT));
                        ee->checkerrors(ERROR_T);
                }

   }
        else
        {       printf("\n%s", t->Thisline());
                printf("Line %d %-80s", t->Line_num(), ee->readerror(_W_STMTN
OEFFEC));
                ee->checkerrors(WARNING_T);
        }

}

/* end of file */
```

```
//---------------------------------[table.h]--------------------------------

#ifndef TABLE_H
#define TABLE_H

#include <stdio.h>
#include "..\head\stmts.h"

#define TSTRING 0
#define TINT    1
#define TBLSZ   20

// define the basic element of the symbol table. The symbol table can hold
// any port_t, device_t and procedure_t object. In this implementation only
// port_t and device_t objects are held in the symbol table.
struct  symbol_t
{
        stmt_t* symbol;                 // table entry
        int type;                                       // statement type - c
urrently only port

        // and device types allowed
        symbol_t* next;                 // pointer to next entry

};

// define symbol table class to store port and device variable info
class table {
        symbol_t** tbl;
        int size;
        unsigned numOfdevs;
        unsigned numOfports;
public:
        table(void);            // not used
        table(int size);        // define table with a specified size
        ~table();
        int getsize() { return (size); }
        unsigned get_numOfdev() { return (numOfdevs); }
        unsigned get_numOfports() { return (numOfports); }

        symbol_t* search(stmt_t*, char*, int, int);
        symbol_t* insert(stmt_t* s, char* c, int t)
                  { return (search(s, c, t, 1)); }  // search inserts the sym
bol if t=0

   // type cast symbol to port_t
        port_t* convertPort(stmt_t* ss) { return ((port_t *)ss); }
        // type cast symbol to device_t
        device_t* convertDevice(stmt_t* ss) { return ((device_t* )ss); }

        symbol_t* getsymbol(int i) { return (tbl[i]); }
        int Type(symbol_t* s) { return (s->type); }    // unnecessary functio
n
};

#endif

/* end of file */
```

```
// --------------------------------table.cpp ---------------------------------
-

#include <string.h>
#include "..\head\table.h"

// methods for table class

/*  ********************************************************************** *
/

/*
   FUNCTION:    creates an instance of the table and allocates space specifie
d
   by the input parameter.

*/

/* ********************************************************************** *
/

table::table(int sz)
{
        if (sz < 0) printf("negative table size");
        tbl = new symbol_t*[size = sz];
        for (int i = 0; i<sz; i++) tbl[i] = NULL;  // initialize table
        numOfdevs = 0; numOfports = 0;
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Deallocates space previously assigned for the table object.

*/

/* ********************************************************************** *
/

table::~table()
{
        for (int i = 0; i<size; i++)
                if (tbl[i])
                {
                        delete tbl[i]->symbol;
                        delete tbl[i];
                }
        delete tbl;
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    This function performs two main operations. It inserts an obj
ect
        into the table if the insert flag is set. Otherwise it searches for t
he
        object in the symbol table and returns a pointer to it.
```

```
*/

/* ************************************************************************** *
/

symbol_t* table::search(stmt_t* st, char* id, int tt, int ins)
{
        port_t* pt;
        device_t* dt;

        int i = 0;
           // search for the string
           for (symbol_t* n=tbl[i]; n && i<size; n=n->next)
           {
                switch (tt)
                {
                   case KPORTS:
                        pt = convertPort(n->symbol);
                        if ((strcmp(id, pt->get_portName()) == 0) &&
                            ( tt == n->type))
                            { n->symbol = pt;
                              return n;  }
                        break;
                   case KDEVICES:
                        dt = convertDevice(n->symbol);
                        if ((strcmp(id, dt->get_deviceName()) == 0) &&
                            ( tt == n->type))
                            { n->symbol = dt;
                              return n;  }
                        break;
                   OTHERWISE:
                        if ((strcmp(id, n->symbol->nameOf()) == 0) &&   \
                            ( tt == n->type)) return n;
                           break;
                }
                i++; // n = n->next;
           }
        // if string not found return error
        if (!n && ins == 0)
        {
                return (NULL);
        }

        symbol_t* tp = new symbol_t;
        // create an entry in the table
        switch (tt)
        {
           case KPORTS:
                        tp->symbol = convertPort(tp->symbol);
                        tp->symbol = new port_t();
                        tp->symbol = convertPort(st);
                        numOfports++;
                        break;
           case KDEVICES:
                        tp->symbol = convertDevice(tp->symbol);
                        tp->symbol = new device_t();
                        tp->symbol = convertDevice(st);
                        numOfdevs++;
                        break;
```

```
            default:
                 break;
        }
        tp->type = tt;
        tp->next = NULL;
        tbl[i-1]->next = tp;
        tbl[i] = tp;
        return (tp);
}

/* end of file */
```

```
//--------------------------------[ token.h ]--------------------------------
-

#ifndef TOKEN_H
#define TOKEN_H

#include "..\head\char.h"

/* Token Types */

#define TILLEGAL        0
#define TIDENTIFIER     1
#define TINTEGER        2
#define TLPAREN         3
#define TRPAREN         4
#define TCOLON          5
#define TSEMICOLON      6
#define TDOT            7
#define TQUOTE          8
#define TEOF            9
#define TCOMMA          10
#define TKEYWORD        11
#define TFUNCTION       12
#define TCHARSTRING     13
#define TSPACE          14

/* Define Keywords */

#define KPORTS          0
#define KEND            1
#define KINPUT          2
#define KOUTPUT         3
#define KDEVICES        4
#define KCOIL           5
#define KSENSOR         6
#define KPULSE          7
#define KPROGRAMMABLE   8
#define KPLAIN          9
#define KON             10
#define KOFF            11
#define KWAITON         12
#define KWAITOFF        13
#define KSEND           14
#define KWAIT           15
#define KSTROBE         16
#define KDO             17
#define KPROCEDURE      18
#define KLPT1           19
#define KCOM1           20
#define KCOM2           21


#define MAX_ID 30           // Maximum identifier size

#define PARPORT 0         // Parallel port type
#define SERPORT 1         // Serial port type
#define ADDRPORT        2                       // Address ports


// define class for token type
```

```
class token_t {
    long int ivalue;         // integer value
    char string[MAX_ID+1];   // identifier value
    int token;                          // token type
    int reget_flag;          // character look ahead flag
    int keytype;                        // keyword type
    int line_count;
         char_t *cp;
         FILE * tp;                                          // temperory file to
 store procedure source code
         int sendsource;                           // flag to indicate source c
ode lines to be sent
    int newline;
public:
         token_t(char_t *c, FILE* t) { reget_flag = 0; line_count = 1; cp = c
;

         sendsource = 0; tp = t; newline = 0; };
         // forms tokens out of characters read from the source file
         int Next(void);
         long int Integer(void) { return(ivalue); };
         char *Identifier(void) { return(string); };
         int Type(void) { return(token); };
         void Reget(void) { reget_flag = 1; };
         int Keytype(void) { return(keytype); };
         int Line_num(void) { return(line_count); };
         void SendSource() { sendsource = 1; };
         void StopSend() { sendsource = 0; };
         int CheckSend() { return (sendsource); };
         char* Thisline() { return (cp->Thisline()); }
};


#endif
```

```
//------------------------------[ token.cpp ]---------------------------------
-

#include <string.h>
#include <iostream.h>
#include "..\head\char.h"
#include "..\head\token.h"
#include "..\head\errors.h"

extern error_t *ee;

/* Define Keywords */
static char *keywords[] = {
        "Ports",                                // KPORTS
        "End",                          // KEND
        "Input",                                // KINPUT
        "Output",                       // KOUTPUT
        "Devices",                      // KDEVICES
        "Coil",                         // KCOIL
        "Sensor",                       // KSENSOR
        "Pulse",                                // KPULSE
        "Programmable", // KPROGRAMMABLE
        "Plain",                                // KPLAIN
        "On",                                   // KON
        "Off",                          // KOFF
        "WaitOn",                       // KWAITON
        "WaitOff",                      // KWAITOFF
        "Send",                         // KSEND
        "Wait",                         // KWAIT
        "Strobe",                       // KSTROBE
        "Do",                                   // KDO
        "Procedure",            // KPROCEDURE
        "LPT1",                         // KLPT1
        "COM1",                         // COM1
        "COM2"                          // COM2
};

/* Scanner States */

#define SINI            0               /* Start (Initial) state */
#define SID             1               /* Identifier */
#define SINT            2               /* Integer */
#define STR                             3                       /* St
ring */
#define SQTE                            4                       /* Quote */
#define SLNF                            5                       /* Linefeed s
tate */
#define SDONE           10              /* final state */

short action_tbl[13][6] = {
            /*  States
                                        S   S   S   S   S   S
                                        I   I   I   T   Q   L
                                        N   D   N   R   T   N
                        I       T       E   F    */
/* CILL    */  0, 12, 14, 17, 19, 20,
/* CWHITE  */  1, 12, 14, 17, 19, 1,
/* CQUOTE  */  2, 12, 14, 18, 17, 20,
/* CID     */  3, 13, 14, 17, 19, 20,
/* CLPAREN */  4, 12, 14, 17, 19, 20,
```

```
/* CRPAREN */  5, 12, 14, 17, 19, 20,
/* CCOMMA  */  6, 12, 14, 17, 19, 20,
/* CDOT    */  7, 12, 14, 17, 19, 20,
/* CDIG    */  8, 13, 15, 17, 19, 20,
/* CCOLON  */  9, 12, 14, 17, 19, 20,
/* CSEMI   */ 10, 12, 14, 17, 19, 20,
/* CLF     */ 11, 12, 14, 20, 19, 11,
/* CEOF    */ 16, 12, 14, 20, 19, 16
};


/*  ********************************************************************** *
/

/*
   FUNCTION:    It scans the source file and classifies characters into
   different tokens

*/

/* ********************************************************************** *
/


int token_t::Next(void)
{
        // initialize the token state table
        int     state = SINI,
        slen = 0;

        // Reset reget flag if set and return token
   if (reget_flag)
        {
                reget_flag = 0;
                return(token);

#ifdef  VERBOSE
                if (token == TINTEGER) cout << ivalue << "\n";
                else cout << string << "\n";
#endif
        }
        do
   {
                cp->next();     // get next character to be parsed
                switch(action_tbl[cp->Class()][state])
                {
                case 0: /* illegal character */
                                token = TILLEGAL;
                                state = SDONE;
                                break;
                case 1: /* white space */
                                break;
                case 2: /* quote */
                                token = TQUOTE;
                                state = STR;
                                break;
                case 3: /* initial identifier character */
                                string[0] = cp->code();
                                slen = 1;
                                state = SID;
```

```
                                break;
                        case 4: /* left paren */
                                token = TLPAREN;
                                state = SDONE;
                                break;
                        case 5: /* right paren */
                                token = TRPAREN;
                                state = SDONE;
                                break;
                        case 6: /* Comma */
                                token = TCOMMA;
                                state = SDONE;
                                break;
                        case 7: /* Dot */
                                token = TDOT;
                                state = SDONE;
                                break;
                        case 8: /* Digit, initial */
                                ivalue = cp->code() - '0';
                                state = SINT;
                                break;
                        case 9: /* Colon */
                                token = TCOLON;
                                state = SDONE;
                                break;
                        case 10: /* Semicolon */
                                token = TSEMICOLON;
                                state = SDONE;
                                break;
                        case 11: /* Line Feed */
                                cp->Readline(); cp->reset_nextchar();
                                line_count++;
                                newline = 1;
//              state = SLNF;
                                break;
                        case 12: /* Identifier state/other character */
                                string[slen] = '\0';
                                cp->reget();
                                token = TIDENTIFIER;

                                // check for keyword
                                for (int i = 0; i < sizeof(keywords)/sizeof(k
eywords[0]); i++)
                                {
                                        if (strcmp(string,keywords[i]) == 0)
                                        {
                                                token = TKEYWORD;
                                                keytype = i;
                                        }
                                }
                                // Check for keyword so that keyword lines ar
e not printed into
            // the trace file
                                if ((token != TKEYWORD) && CheckSend() && (ne
wline))
                                        fprintf(tp, "%s", cp->Nextline());
            if (newline) newline = 0;
                                state = SDONE;
            break;
                        case 13: /* Identifier state/valid character */
```

```
                                if (slen < MAX_ID ) string[slen++] = cp->code
();
                                break;
                        case 14: /* Integer state/other character */
                                token = TINTEGER;
                                state = SDONE;
                                cp->reget();
                                break;
                        case 15: /* Integer state/valid character (digit) */
                                ivalue = (ivalue * 10) + (cp->code() - '0');
                                break;
                        case 16: /* End-of-file */
                                token = TEOF;
                                state = SDONE;
                                break;
                        case 17:  /* String initial */
                                string[slen++] = cp->code();
                                break;
                        case 18:  /* Quote encountered in String */
                                state = SQTE;
                                break;
                        case 19: // quote state other character
                                string[slen] = '\0';
                                cp->reget();
                                token = TCHARSTRING;
                                state = SDONE;
                                break;
                        case 20:
                                line_count++;
                                cp->reget();
                                newline = 1;
                                state = SLNF;
            break;
                        default:
                                printf("%s\n", ee->readerror(_F_PARSERERROR))
;
                                ee->checkerrors(FATAL_T);
                                break;
                }
         } while (state != SDONE);

#ifdef  VERBOSE
        if (token == TINTEGER) cout << ivalue << "\n";
        else cout << string << "\n";
#endif

        return(token);
}

/* end of file */
```

```
//--------------------------------[ trans.h ]---------------------------------
-

#ifndef TRANS_H
#define TRANS_H

#define SETBIT                  1
#define RESETBIT                0
#define MAXSTR                  80
#define MAXID                   30
#define EXTDELIMIT      '.'
#define BACKSLASH               '\\'
#define OUTEXT                  ".out"
#define REFEXT                  ".ref"
#define CMDEXT                  ".cmd"
#define TRACEXT         ".trc"


#include <stdio.h>
#include <string.h>

#include "..\head\char.h"
#include "..\head\token.h"
#include "..\head\stmts.h"
#include "..\head\list.h"
#include "..\head\crossref.h"
#include "..\head\errors.h"

extern list_t* procPtr;     // list class defined in list.h
extern char* strtrn(char* , const char);  // to be included in system library
extern error_t * ee;

// translator class
class translate_t {
    int stmt_ct;                //number of executable statements
    int error_ct;               // number of errors
    int warning_ct;     // number of warning errors
    token_t * t;
         char_t * c;
    char directory[MAXID];
         crossref_t cc;      // cross reference object
         char outfile[MAXID], refile[MAXID], tracefile[MAXID];
         FILE *pp;           // source file pointer
    FILE *trp;                  // temperory file pointer

public:
         translate_t(char* f)
         {
                        char *temp, *p;

                        stmt_ct=0;                                  // number of
lines compiled
                        error_ct=0;                                 // number of
compile errors
                        warning_ct=0;                       // number of warning
errors

                        if ((pp = fopen(f, "r")) == NULL)
         {
                                printf("%s\n", ee->readerror(_F_UNOPNSRCFIL)
```

```
);
                                    ee->checkerrors(FATAL_T);
        }
                        else
                        {       strcpy(directory, f);
                                p = strrchr(directory, BACKSLASH);  // retain
 only the directory
                                *(++p) = '\0';

                                temp = new char[strlen(f)+1];
                                strcpy(temp, f);

                                // strip off the source file extension
                                *(strrchr(temp,EXTDELIMIT)) = '\0';
                                strcat(strcpy(outfile, temp), OUTEXT);
                                strcat(strcpy(refile, temp), REFEXT);
            strcat(strcpy(tracefile, temp), TRACEXT);

#ifdef VERBOSE
                                printf("ouput file is %s\n", outfile);
                                printf("cross ref file is %s\n", refile);
#endif
                                c = new char_t(pp);
                                trp = fopen(tracefile, "w");
                                t = new token_t(c,trp);
                                procPtr = new list_t;

                        }
        }

        int parsePorts(void);
        int parseDevices(void);
    int parseProcedure(void);
        int generate(void);
        int crossrefer(void);
        FILE* Tempfile() { return (trp); }
    friend procedure_t::procedure_t(token_t* );
};

#endif
```

```
//------------------------------[trans.cpp]---------------------------------
-


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "..\head\functbl.h"
#include "..\head\trans.h"
#include "..\head\stmts.h"
#include "..\head\table.h"
#include "..\head\list.h"
#include "..\head\crossref.h"
#include "..\head\errors.h"

extern table* sytb;
extern list_t* procPtr;
extern tbltype functbl;
extern int numports;
extern error_t* ee;

// methods for translate object

/*  ************************************************************************ *
/

/*
   FUNCTION:    Begins to parse port declaration statements when the keyword
   PORTS is encountered. The parsing is continued by the port object when it
   is instantiated and inserted into the symbol table.

*/

/* ************************************************************************ *
/

int translate_t::parsePorts(void)
{
    int result = 0;

#ifdef VERBOSE
         printf("Parse Ports\n");
#endif

    result = ((t->Next() == TKEYWORD) &&  // check for keyword PORT
              (t->Keytype() == KPORTS));
    if (!result)   // if not generate error
         {
                printf("\n%s", t->Thisline());
                printf("Line %d %s\n", t->Line_num(), ee->readerror(_E_PORTKE
YEXPT));
                ee->checkerrors(ERROR_T);
         }
         else
         {
                // get all port declarations
      t->SendSource();
                while (!((t->Next() == TKEYWORD) && (t->Keytype() == KEND)))
                // make port entry into symbol table
```

```
                        sytb->insert(new port_t(t), "", KPORTS);
          }
          t->StopSend();
     return(result);
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Begins to parse device declaration statements when the keywor
d
   DEVICE is encountered. The parsing is continued by the device object when
   it is instantiated and inserted into the symbol table.

*/

/* ********************************************************************** *
/

int translate_t::parseDevices(void)
{
     int result = 0;

#ifdef VERBOSE
          printf("Parse Devices\n");
#endif

     result = ((t->Next() == TKEYWORD) &&  // check for keyword DEVICES
               (t->Keytype() == KDEVICES));
     if (!result)     // if not generate error
          {
                printf("%s", t->Thisline());
                printf("Line %d %s\n", t->Line_num(), ee->readerror(_E_DEVKEY
WEXPT));
                ee->checkerrors(ERROR_T);
          }

          else
       // get all device declarations
                while (!((t->Next() == TKEYWORD) && (t->Keytype() == KEND)))
                        // make device entry into the symbol table
                        sytb->insert(new device_t(t), "", KDEVICES);
     return(result);

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Begins to parse procedure statements when the keyword
   PROCEDURE is encountered. The parsing is continued by the procedure object
   when it is instantiated and inserted into the symbol table.

*/

/* ********************************************************************** *
/
```

```
int translate_t::parseProcedure(void)
{
    int result = 0;

#ifdef VERBOSE
         printf("Parse Procedure\n");
#endif

        if (!feof(pp))
        {
                result = ((t->Next() == TKEYWORD) &&  // check for keyword PR
OCEDURE
                         (t->Keytype() == KPROCEDURE));
                if (!result)      // if not generate error
                {
                        printf("%s", t->Thisline());
                        printf("Line %d %s\n", t->Line_num(), ee->readerror(_
E_PROCKEYEXPT));
                        ee->checkerrors(ERROR_T);
                }

                else
                        // get all port declarations
      {
         t->SendSource();
                        while (!((t->Next() == TKEYWORD) && (t->Keytype() ==
KEND)))
                        // make port entry into symbol table
                                procPtr->append(new procedure_t(t));
      }
        }
        t->StopSend();
        fclose(trp);
        return(result);

}

/*  ********************************************************************** *
/

/*
    FUNCTION:   generates p-code for the functions in the procedure declarati
on.
    The p-code is output into a text file, which serves as input to the
    interpreter.

*/

/* ********************************************************************** *
/

int translate_t::generate(void)
{
   FILE* fp;
        FILE * rp, temp;
   char sendstr[MAXSTR], directionStr[MAXSTR];
   objptr_t code;
        long curpos;
   int len;
        char cmdfile[MAXID];
```

```
        symbol_t* ps;                    // to hold port symbols from the symb
ol table
        port_t* pt;                                     // port type pointer

#ifdef VERBOSE
     printf("Generate code \n");
#endif

        code = procPtr->first();

   // create output file with same filename as input file but with an
   // extension of 'out'
        if ((fp = fopen(outfile, "w")) != NULL)
        {
         if ((trp = fopen(tracefile, "r")) != NULL)
         {
                // generate pseudo opcodes for port declarations
                for (int i=0; i<numports; i++)
                {
                        fprintf(fp, "%d %s", CSOURCE, fgets(sendstr, MAXSTR,
trp));
                        // search for identifier in symbol table
                        if ((ps = sytb->getsymbol(i)) != NULL)
         {
                                switch (ps->type)
                                {
                                        case KPORTS:
                pt = sytb->convertPort(ps->symbol);
                                                switch (pt->get_type())
                                                {
                                                        case ADDRPORT:
                                                                switch (pt->g
et_direction())// Set up direciton string
                                                                {       case
KINPUT: strcpy(directionStr,"INPUT"); break;
                                                                        case
KOUTPUT: strcpy(directionStr,"OUTPUT"); break;
                                                                        defau
lt: break;
                                                                }
                                                                fprintf(fp, "
%d %u %s\n", CPORT, pt->get_portAddress(),
        directionStr);  // direction as a string
                        break;
                                                        case SERPORT:
                                                                fprintf(fp, "
%d %d %d %d %d\n", CSERPORT, pt->Baudrate(),
                                                                        pt->S
tartBit(), pt->StopBit(), pt->ParityBit());
                                                                break;
                                                        default:
                                                                break;
                                                }
                                                break;
                                        default:
                                                break;
            }
                        }
                }
```

```
     // generate pseudo opcodes for functions
     while ((code != NULL))
          {
                         fprintf(fp, "%d %s", CSOURCE, fgets(sendstr, MAXSTR,
trp));

                         switch (code->codes.opcode)
                         {
                                 case CON:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case COFF:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                         break;
                                 case CWAITON:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case CWAITOFF:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                         break;
                                 case CSEND:
                                         fprintf(fp, "%d %s %s\n", code->codes
.opcode, code->codes.comport, code->codes.string );
                                         break;
                                 case CWAIT:
                                         fprintf(fp, "%d %d\n", code->codes.op
code, code->codes.waitsecs);
                                         break;
                                 case CSTROBE:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case CDO:
                  // command files are stored in the user's directory
                                         strcpy(cmdfile, directory);
                 // append "cmd" extension for command files
                                         strcat(cmdfile, code->codes.filename)
;
                                         if ((rp = fopen(strcat(cmdfile, CMDEX
T), "r")) != NULL)
                                         {
                                                 while (!feof(rp))
                                                 {
                                                         if (fgets(sendstr, MA
XSTR, rp) != NULL)
                                                         fprintf(fp, "%d %s %s
", code->codes.opcode, code->codes.comport, sendstr );
                                                 }
                                                 fclose(rp);
                                         }
                                         else
                                         {       printf("%s %s\n", ee->readerr
or(_F_UNOPNCMDFIL), code->codes.filename);
                                                 ee->checkerrors(FATAL_T);
                  }
```

```
                                        break;
                                default:
                                        break;
                        }
                        code = (procedure_t*) procPtr->next();
                }
        }
        else
        {       printf("%s\n", ee->readerror(_W_UNOPNTRCFIL));
                ee->checkerrors(WARNING_T);
     }
     fclose (trp);
        fclose (fp);
      }
      else
      {       printf("%s\n", ee->readerror(_F_UNOPNOUTFIL));
              ee->checkerrors(FATAL_T);
              return(0);
      }
      fclose(pp);
      return (1);
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Builds and prints the cross reference table.  Refer to
    crossref::build for more information.

*/

/* ********************************************************************** *
/

int translate_t::crossrefer(void)
{
        typedef int ndxarr[10];

        ndxarr i1, i2;       // indices into the cross reference table

        cc.build(i1, i2);            // build cross references
        cc.print(refile, i1, i2);    // print into a separate file
        cc.display(refile);          // display cross references
        return (1);
}

/* end of file */
```

```
100 %_F_UNOPNSRCFIL Fatal error - Unable to open source file
101 %_F_UNOPNERRFIL Fatal error - Unable to open errors data
102 %_F_UNOPNOUTFIL Fatal error - Unable to create output file
103 %_F_EREADSRCFIL Fatal error - Error reading source file
104 %_F_PARSERERROR Parser error - undefined action code
105 %_F_UNOPNCMDFIL Fatal error - Unable to open command file
106 %_F_UNREAERRFIL Fatal error - Unable to read from error file
107 %_F_UNRESERRPTR Fatal error - Unable to reset error file pointer
111 %_E_SEMICOLEXPT Syntax error - Semicolon ; expected
112 %_E_FUNCOPREXPT Syntax error - funciton operator expected
113 %_E_UNDEFIDENTF Syntax error - Undefined identifier
114 %_E_IDENTIFEXPT Syntax error - Identifier expected
115 %_E_TYPADDREXPT Syntax error - Type name or address expected
116 %_E_INCORSERPOR Syntax error - Incorrect serial port
117 %_E_INTEGEREXPT Syntax error - Integer expected
118 %_E_INVALDEVTYP Syntax error - Invalid device type
119 %_E_DEVTYPEXPTD Syntax error - device type expected
120 %_E_INVALIDFUNC Syntax error - Invalid function name
121 %_E_KEYWORDEXPT Syntax error - keyword expected
122 %_E_PARTYPMISMA Syntax error - parameter type mismatch
123 %_E_PORTKEYEXPT Syntax error - expected keyword: Ports
124 %_E_DEVKEYWEXPT Syntax error - expected keyword: Devices
125 %_E_PROCKEYEXPT Syntax error - expected keyword: Procedure
150 %_W_UNOPNCRFILE Warning - Error opening cross reference file
151 %_W_BAUDNOTSPEC Warning - Baudrate not specified
152 %_W_STMTNOEFFEC Warning - Statement has no effect in code
153 %_W_UNOPNTRCFIL Warning - Unable to open trace file
```

```
* Port Declarations
Ports
        PortC 64259 Output;
        PortA 64256 Input;
     Com1port COM1 3600 3 2 1;
End

* Device Declarations
Devices
        PalletLiftUp Pulse PortC 4;
        Conveyor Coil PortC 5;
        PhotoCell Sensor PortA 7;
        PalletArrived Sensor    PortA   6;
        ChuckOpen Pulse PortC 1;
        Robot Programmable LPT1;
        Lathe Programmable Com1port;        * com port declared in port secti
on
        LatheStart Pulse PortC 2;
        LatheStop Sensor PortA 4;
        PalletLifted Sensor PortA 5;
        PalletStops Coil PortC 0;
        ChuckClose Pulse PortC 3;
        PalletLiftDown Pulse PortC 6;
        LatheRunning Sensor PortA 2;
        LatheHandShk Sensor PortA 3;
        Delay Wait;
End

* Procedure Declarations
Procedure
        Lathe.Do(loadlathe);
        Robot.Send("NT");
        PalletStops.On;
        Conveyor.On;
        PhotoCell.WaitOn;
        PalletStops.Off;
        PalletArrived.WaitOn;           * Wait on part arrival
        Delay.1000;
        PalletLiftUp.Strobe;
        PalletLifted.WaitOn;
        Conveyor.Off;
        ChuckOpen.Strobe;
        Robot.Do(loadpart);
        Delay.1000;
        ChuckClose.Strobe;
        Delay.2000;
        Robot.Do(moveaway);
        Delay.2000;
        LatheStart.Strobe;
        LatheStop.WaitOff;
        Robot.Do(MoveBack);
        Delay.2000;
        ChuckOpen.Strobe;
        Delay.2000;
        Robot.Do(GetPart);
        PalletStops.On;
        PalletLiftDown.Strobe;
        Conveyor.On;
        Delay.500;
        Conveyor.Off;
```

```
        LatheStart.Strobe;
        PalletStops.Off;
End
```

```
* Port Declarations
Ports
        PortA 64256 Input;
        PortC 64257 Output;
        Com1port COM1 3600 3 2 1
End

* Device Declarations
Devices
        PalletLiftUp Pulse PortC 4;     * Pulse device
        Conveyor Coil PortC 5;
        PhotoCell Sensor PortA 7;
        Lathe Programmable Com1port;
        Robot Programmable LPT1;
End

* Procedure Declarations
Procedure
        Robot.Send("NO");
        PalletLiftUp.Strobe;

        * Turn the conveyor on
        Conveyor.On;

        PhotoCell.WaitOn;
        Conveyor.Off;
        PalletLiftUp.Strobe;
   Lathe.Send("NT");
End
```

APPENDIX H: CPL Language Reference & User Manual

1. Introduction

The Cell Programming Language (CPL) is a high-level special purpose language being developed at the Department of Systems Analysis at Miami University. This project is part of a larger project to design a computer aided manufacturing system, and support course work, projects, and research in Flexible Manufacturing.

1.1. What is CPL

CPL is a programming language environment for use in the control of manufacturing cells. Individual cell components and their operations can be integrated by programming the cell as a single unit. Programs to do this could be written in any other existing high-level language such as BASIC or C, but the user would have to be familiar with the syntax necessary to perform low-level input and output to the various hardware devices that provide the interface to the cell's devices. For example, the user would set a particular bit on a particular hardware port to 1 to turn on a device. Instead, CPL allows the user to program the cell by using commands such as On and Off, and the CPL system will take care of the low-level programming details.

CPL does not hide all of the hardware details. In order to use CPL, the user is still required to know the particular hardware device and bit to which each device is interfaced. Also, the user must know the type of device. Finally, individual cell components such as robots and CNC machines will have to be programmed in their host languages. One advantage, however, is that the programmer has full control of the operations, and can communicate with the individual devices even after the programs have been loaded into their memories.

1.2. The CPL Environment

The CPL environment consists of :

1. CAD/CAM workstations, a file server and peripherals;
2. A local area network;
3. Personal Computer (PC) controlled manufacturing cells;
4. Interfacing electronics between the PCs and the cell devices;
5. A programming language used to program the cells.

An overview of the environment can be found in the paper "Object-Oriented Flexible Manufacturing System at Miami University" in Appendix A of this document. The remainder of this document is devoted to the description of item 5 in the above list.

2. How CPL Works?

The CPL software consists of three major components:

1. CPL compiler;

2. CPL interpreter;
3. Remote status display.

The CPL compiler processes the user's CPL program along with any required robot and/or CNC command files to produce an intermediate file of instructions known as p-code. This p-code is the input to the interpreter which performs the low-level input/output operations on the cell controller PC. Thus, the compiler can be run on any PC or CAD/CAM workstation, but the interpreter must reside on the cell controlling computer. Once a CPL project has been compiled to p-code, it need not be recompiled unless a change is made in the CPL code. The machine command files serve only as input so any changes made to them will not affect the execution of the CPL program. A debugging option is provided in the interpreter which helps to eliminate CPL program errors.

The remote status display is an optional component of the system that can be used to remotely monitor the operation of the cell. The remote status display has a component, called the monitor, that runs on the cell controller, and a component, called the display, that runs on a remote PC, for example a CAD/CAM workstation. The monitor sends the state of each device to the display which in turn outputs the status to the user.

3. A CPL Project

Earlier it was stated that CPL is a language that allows the user to control and integrate devices of a manufacturing cell, but that the user is still required to program individual cells in their host languages. Thus, a CPL program would consist of:

1. A CPL program, and
2. Zero or more command files for programmable devices.

In managing a project, the user should keep all of the command files and the associated CPL language file together, ideally in a separate directory on a CAD/CAM workstation.

4. A CPL Program

A CPL program consists of five major sections: port declarations, device declarations, cell declarations, procedure declarations, and program declarations. The following subsections elaborate on each of these.

4.1. Objects and Classes in CPL

A CPL program has three major types

1. Ports: Used to name hardware interface ports.
2. Devices: Used to name individual cell devices, and assign ports and bit numbers
3. Cells: Used to declare network addresses of cell controlling computer.

In CPL, all data types are classes and the instantiation of an object to be of a type or class

assigns values to the attributes of the of the object. However, a dynamic change to the value of an attribute is not possible.

4.1.1. Port Declarations

The port declaration section is used to assign a physical port address on the PC. The declarations are made within a PORTS.... END block. Following the keyword PORTS is a series of individual port declarations. The syntax of a port declaration is as follows

< port_variable > (< port_address > < direction >)|(< port_name > < baudrate > < data_bits > < stop_bits > < parity >);

The *port_variable* can be any user defined identifier consisting of a maximum of 31 characters. The identifier can consist of alphabetic characters, digits and underscores upto a maximum of 31 characters. The *port_address* should be a physical port address and the *direction* is either INPUT or OUTPUT depending on whether the port is used to send or receive signals; the default direction is INPUT. If the port is a serial port, the *port_name* should be one of the serial ports COM1: or COM2: followed by the *baudrate*, number of *data bits*, number of *stop bits*, and the type of *parity* should be specified. An example port declaration section is given below.

```
Ports
        PortA  64259  Output;
        PortB  64256  Input;
        PortC  64257;
        Com1port COM1: 3600 7 1 1;
End
```

4.1.2. Device Declarations

The device declaration section is used to declare a device object and associate a port and bit number with it. The device types are predefined and correspond to the devices in the cell. We have not made provisions for including user-defined device types in the language, because at this juncture we do not anticipate such a need. The declaration block is bounded by the keywords DEVICES and END. The syntax for the device declaration is as follows,

< device_variable > < device_type > (< port_variable > [< bit_number >]) | < programmable_port > ;

The *device_variable* is a user defined identifier and the *device_type* is a keyword in the language. The *port_variable* should have been defined earlier in the port declaration section, and the *bit_number* is a constant between 0 and 7 and corresponds to a bit on the data acquisition board. For a programmable device type, the port name LPT1 is specified if a parallel port is to be used, and the port identifier that has been assigned one of the serial ports COM1: or COM2: is specified if a serial port is to be used. An example device declaration section is given below.

```
Devices
 PalletLiftup    Pulse    PortC  4;
```

```
Conveyor    Coil    PortC 5;
Robot       Programmable LPT1;
Lathe       Programmable Com1port;
End
```

4.1.3. Cell Declarations

The cell declaration section is used to assign network addresses to cell names in order to provide for communication between cells. Declaring cell names makes it convenient to assign a procedure to a cell, and facilitates modular programming at a small scale. The syntax of a cell declaration is as follows:

< cell_variable > < network_address >;

The *cell_variable* is like any other user defined identifier, and cannot exceed a maximum of 31 characters. The *network_address* is a pre-defined host name or network address that has been assigned to the cell controlling computer by the system administrator. An example of a cell declaration section is given below.

```
Cells
        ManufacCell     cimlab6;
        StorageCell     192.34.54.3;
End
```

4.2. Control Constructs in CPL

A CPL program has two basic control constructs:

7. Procedures: Contains the sequence of cell control operations executed on devices
8. Program: Collection of procedures to be executed on cells

4.2.1. Procedure Declarations

The next section in the program is the procedure section which consists of statement constructs. Each statement represents one device operation and directly corresponds to an actual operation of the real device. The syntax of a procedure statement is as follows,

< device_variable > . (< device_function > [< open_parenthesis > parameter { , . . . }. < close_parenthesis >]) |
< delay_time >

The *device_variable* is an identifier previously declared in the device declaration section. The *device_function* is predefined, and is a keyword in the language. Table 4.2.1 lists device types and valid functions for each device type. Function parameters are enclosed within parenthesis and are separated by commas. As with devices and ports, the keywords PROCEDURE and END mark the beginning and end of a procedure block. An example of the procedure section is given below.

```
Procedure
        Conveyor.On;
        Robot.Send("NT");
```

```
        Lathe.Do(Cutpart);
        Delay.1000;
End
```

TABLE 2.1

TYPES	VALID FUNCTIONS
COIL	ON, OFF
SENSOR	WAITON, WAITOFF
PULSE	STROBE
PROGRAMMABLE	SEND, DO
DELAY	MILLISECONDS

Students can program device objects with names that directly correspond to their real-world counterparts. The predefined device functions are named after the actual device operations. For e.g., a statement such as Conveyor.On is an instruction to switch on the conveyor and a statement such as PhotoCell.WaitOn is an instruction to wait for the photocell to be switched on. This way it is possible to write a program and visualize an entire production operation without actually performing it.

The user is, however, required to program a programmable device type using its host language. Commands to the PROGRAMMABLE device type can be given directly by passing them as parameters to a function or they can be stored in a separate file, and the file name passed as the parameter. For e.g., Robot is declared to be a programmable device, and the Send operation accepts a parameter which is a string and sends a command to the robot. The Do function on the other hand accepts an identifier that is the name of a file consisting of robot commands which are read and directly output to the robot.

4.2.2. Program Declaration

The program section is the last section in a CPL program, and is simply a series of statements arranged in a predetermined order by the programmer. Each procedure statement states what procedure is going to be executed on what cell, and specifies repetition clauses, if any. The program statements appear within a PROGRAM....END block. The syntax of a program statement is as follows:

< cell_name >. < procedure_name > [(< condition > | < repetition_times >)];

The *procedure_name* is the name of a procedure that has been defined previously. Similarly the *cell_name* is also the name of a cell that has already been declared. Repetition clauses, if any, must be specified either as a *condition,* or as the number of *repetition_times* that a proce-

dure is to be executed. These are enclosed within parentheses. An example program declaration is given below.

```
Program
        ManufacCell.ProduceBody;
        StorageCell.StoreParts (ManufacCell.SignalOn);
        ManufacCell.ProduceBody(50);  * Repeat 50 times
        StorageCell.StoreParts(50);
End
```

In CPL, every statement excepting block markers end with a semicolon. The programmer may insert comments in the program by preceding the comment with an asterisk, which is the comment character. A valid program statement and a comment may be typed on the same line, but the comment should follow the program statement. The reverse is, however, not true, because all characters in a line following a comment character are ignored by the CPL compiler. The syntax of this language is kept simple and compact in order to make it more appealing to users. At the end of the compilation, a cross-reference output listing the cross referencing between various devices and ports is printed. Error handling is performed by an error object which prints out error messages with corresponding line numbers and error codes. An example of a complete CPL program is given in appendix E.

5. Using CPL

To invoke the CPL system from the network, type the command

CPL *drive pathname filename*

The *drive* is the drive on which the input and program files are located, and the *pathname* is any absolute or relative DOS path specification. Absolute path specifications are given from the root directory and relative path names are given from the current directory. The *filename* is any combination of alphabetic characters, and the entire file specification should not exceed thirty characters.

The CPL program file can be edited using any standard editor, and should be stored as an ASCII text file. One advantage of structuring the CPL system as two independent components is that the execution is not tied up with the compilation process. To execute the CPL, program type the command

CIMINT *drive:pathname filename.*

The CPL program can also be executed in the debug mode. To do this, type the command

CIMINT –d *drive pathname filename.*

The debug mode will execute the program in steps and the user can trace through the program. This is especially useful for locating program errors.

6. CPL Errors

Errors in the program are detected by the compiler and displayed onto the screen at the end of the compilation. At this point errors are not output into a list file, but this feature may appear in future versions of the compiler. CPL compiler errors are of three types: Syntax Errors, Fatal Errors, and Warning Errors. Currently the compiler allows a maximum of five syntax errors before it terminates abnormally. Fatal errors are those that make it impossible for the compilation process to continue. As such the compiler terminates execution immediately after a fatal error is discovered; no count of fatal errors is kept. Warning errors are those that do not seriously affect the production of p-code, but may produce unpredictable results at run-time, or impede the debugging process by producing incorrect cross references. Although warning errors are detected and displayed, they are ignored by the compiler. A description of the standard error messages produced by the CPL compiler are listed in the following pages, along with hints to rectify the errors.

6.1. FATAL Errors

UNOPNSRCFIL - Unable to open source file

The compiler was unable to open the source file, because of incorrect path specification or access violation. Restart compiler with correct path name, and check access protection for the file.

UNOPNERRFIL - Unable to open errors database

The compiler was unable to open the file containing the compiler error messages. Consult the system administrator or the person in charge of maintaining the CPL system.

UNOPNOUTFIL - Unable to create output file

The compiler was unable to create the output p-code file. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name.

EREADSRCFIL - Unable to read source file

The source program file contained some extraneous characters which the program could not decipher, or the compiler does not have read access for the file. Check the source file for any extraneous characters, and also check the group and world access protection for the file.

PARSERERROR - Undefined action code

A bug in the CPL compiler. Consult the system administrator.

UNOPNCMDFIL – Unable to open the command file

The compiler was unable to open a command file specified in the program. Check the name of the command file specified for spelling errors, and also check if the command file is present in the same directory as the input file.

UNREAERRFIL – Unable to read from errors file

The compiler was unable to read the file containing the error messages. Consult the system administrator.

UNRESERRPTR – Unable to reset error file pointer

The compiler is unable to reset the file pointer for the error file. Consult the system administrator.

6.2. SYNTAX Errors

SEMICOLEXPT – Semicolon ; expected

A semicolon was not found where expected. Insert a semicolon at the end of the statement appearing on the line indicated by the line number in the error message.

FUNCOPREXPT – Function operator expected

The function operator '.' was omitted in a procedure statement. Insert the function operator after the device name in the procedure statement appearing on the line indicated by the line number in the error message.

UNDEFIDENTF – Undefined identifier

The identifier discovered was not declared previously in a declaration section. Enter a declaration for the identifier in the corresponding declaration section.

IDENTIFEXPT – Identifier expected

The compiler was expecting to find an identifier, but could not find one. Include the necessary identifier in the statement appearing on the line indicated by the line number in the error message.

TYPADDREXPT – Type name or address expected

A port variable was not assigned an address or a serial port type. Insert a port address or serial port name in the statement appearing on the line indicated by the line number in the error message.

INCORRSERPORT – An incorrect serial port was specified

An incorrect serial port was specified in the declaration of a serial port variable. Check the serial port name in the definition for predefined serial ports, and correct spelling errors, if any.

INTEGEREXPT – Integer expected

An integer was not found where expected. Check program to locate actual error and make the necessary changes.

INVALDEVTYP – Invalid device type

The device type specified in a device declaration is invalid. Check declaration for spelling errors, and change it to a known device type. If error persists even after correction, consult the system administrator.

DEVTYPEXPT – Device type expected

The compiler was expecting a device type, but did not find one. Check the syntax and make necessary changes.

INVALIDFUNC – Invalid function name

The function name specified is not valid for the accompanying device type. Check the manual for permitted functions for the device type specified, and make the necessary changes.

KEYWORDEXPT – Keyword expected

An was not found where expected. Check program to locate actual error and make the necessary changes. If error persists even after correction, consult the system administrator.

PARITYMISMA – Parameters type mismatch

The parameter type specified does not match with the type expected.

PORTKEYEXPT – expected keywords: Ports

The port declaration section did not begin with the keyword, Ports.

DEVKEYWEXPT – expected keywords: Devices

The device declaration section did not begin with the keyword, Devices.

PROCKEYEXPT – expected keywords: Ports

The port declaration section did not begin with the keyword, Procedure.

6.3. WARNING ERRORS

UNOPNCRFILE – Unable to open cross reference file

The compiler was unable to create or open the cross-reference file. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name. This error does not stop the generation of p-code in the output file, but does not produce useful debugging information.

BAUDNOTSPEC – Baud rate not specified

The baud rate for the serial port specification was not specified in the port declaration section. The compiler will assume the default baud rate which may not match the actual baud rate for the connected device.

STMTNOEFFEC – Statement has no effect in code

A procedure statement or program statement was discovered without any function name. Such a statement is meaningless, and does not produce any p–code.

UNOPNTRCFIL – Unable to open trace file

The compiler was unable to create or open the trace file, which is a temporary file used to output source code statements needed to send trace information to the interpreter. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name. This error does not stop the generation of p–code in the output file, but does not produce useful debugging information.

```
//---------------------------------[table.h]--------------------------------

#ifndef TABLE_H
#define TABLE_H

#include <stdio.h>
#include "..\head\stmts.h"

#define TSTRING 0
#define TINT    1
#define TBLSZ   20

// define the basic element of the symbol table. The symbol table can hold
// any port_t, device_t and procedure_t object. In this implementation only
// port_t and device_t objects are held in the symbol table.
struct  symbol_t
{
        stmt_t* symbol;                 // table entry
        int type;                                       // statement type - c
urrently only port

        // and device types allowed
        symbol_t* next;                 // pointer to next entry

};

// define symbol table class to store port and device variable info
class table {
        symbol_t** tbl;
        int size;
        unsigned numOfdevs;
        unsigned numOfports;
public:
        table(void);            // not used
        table(int size);        // define table with a specified size
        ~table();
        int getsize() { return (size); }
        unsigned get_numOfdev() { return (numOfdevs); }
        unsigned get_numOfports() { return (numOfports); }

        symbol_t* search(stmt_t*, char*, int, int);
        symbol_t* insert(stmt_t* s, char* c, int t)
                  { return (search(s, c, t, 1)); }  // search inserts the sym
bol if t=0

   // type cast symbol to port_t
        port_t* convertPort(stmt_t* ss) { return ((port_t *)ss); }
        // type cast symbol to device_t
        device_t* convertDevice(stmt_t* ss) { return ((device_t* )ss); }

        symbol_t* getsymbol(int i) { return (tbl[i]); }
        int Type(symbol_t* s) { return (s->type); }    // unnecessary functio
n
};

#endif

/* end of file */
```

```
// --------------------------------table.cpp ---------------------------------
-

#include <string.h>
#include "..\head\table.h"

// methods for table class

/*  ********************************************************************** *
/

/*
   FUNCTION:    creates an instance of the table and allocates space specifie
d
   by the input parameter.

*/

/* ********************************************************************** *
/

table::table(int sz)
{
        if (sz < 0) printf("negative table size");
        tbl = new symbol_t*[size = sz];
        for (int i = 0; i<sz; i++) tbl[i] = NULL;  // initialize table
        numOfdevs = 0; numOfports = 0;
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Deallocates space previously assigned for the table object.

*/

/* ********************************************************************** *
/

table::~table()
{
        for (int i = 0; i<size; i++)
                if (tbl[i])
                {
                        delete tbl[i]->symbol;
                        delete tbl[i];
                }
        delete tbl;
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    This function performs two main operations. It inserts an obj
ect
        into the table if the insert flag is set. Otherwise it searches for t
he
        object in the symbol table and returns a pointer to it.
```

```
*/

/* ************************************************************************** *
/

symbol_t* table::search(stmt_t* st, char* id, int tt, int ins)
{
        port_t* pt;
        device_t* dt;

        int i = 0;
           // search for the string
           for (symbol_t* n=tbl[i]; n && i<size; n=n->next)
           {
                switch (tt)
                {
                   case KPORTS:
                        pt = convertPort(n->symbol);
                        if ((strcmp(id, pt->get_portName()) == 0) &&
                            ( tt == n->type))
                            { n->symbol = pt;
                              return n;  }
                        break;
                   case KDEVICES:
                        dt = convertDevice(n->symbol);
                        if ((strcmp(id, dt->get_deviceName()) == 0) &&
                            ( tt == n->type))
                            { n->symbol = dt;
                              return n;  }
                        break;
                   OTHERWISE:
                        if ((strcmp(id, n->symbol->nameOf()) == 0) &&   \
                            ( tt == n->type)) return n;
                           break;
                }
                i++; // n = n->next;
           }
        // if string not found return error
        if (!n && ins == 0)
        {
                return (NULL);
        }

        symbol_t* tp = new symbol_t;
        // create an entry in the table
        switch (tt)
        {
           case KPORTS:
                        tp->symbol = convertPort(tp->symbol);
                        tp->symbol = new port_t();
                        tp->symbol = convertPort(st);
                        numOfports++;
                        break;
           case KDEVICES:
                        tp->symbol = convertDevice(tp->symbol);
                        tp->symbol = new device_t();
                        tp->symbol = convertDevice(st);
                        numOfdevs++;
                        break;
```

```
            default:
                 break;
        }
        tp->type = tt;
        tp->next = NULL;
        tbl[i-1]->next = tp;
        tbl[i] = tp;
        return (tp);
}

/* end of file */
```

```
//--------------------------------[ token.h ]--------------------------------
-

#ifndef TOKEN_H
#define TOKEN_H

#include "..\head\char.h"

/* Token Types */

#define TILLEGAL        0
#define TIDENTIFIER     1
#define TINTEGER        2
#define TLPAREN         3
#define TRPAREN         4
#define TCOLON          5
#define TSEMICOLON      6
#define TDOT            7
#define TQUOTE          8
#define TEOF            9
#define TCOMMA          10
#define TKEYWORD        11
#define TFUNCTION       12
#define TCHARSTRING     13
#define TSPACE          14

/* Define Keywords */

#define KPORTS          0
#define KEND            1
#define KINPUT          2
#define KOUTPUT         3
#define KDEVICES        4
#define KCOIL           5
#define KSENSOR         6
#define KPULSE          7
#define KPROGRAMMABLE   8
#define KPLAIN          9
#define KON             10
#define KOFF            11
#define KWAITON         12
#define KWAITOFF        13
#define KSEND           14
#define KWAIT           15
#define KSTROBE         16
#define KDO             17
#define KPROCEDURE      18
#define KLPT1           19
#define KCOM1           20
#define KCOM2           21


#define MAX_ID 30           // Maximum identifier size

#define PARPORT 0         // Parallel port type
#define SERPORT 1         // Serial port type
#define ADDRPORT        2                       // Address ports


// define class for token type
```

```
class token_t {
    long int ivalue;         // integer value
    char string[MAX_ID+1];   // identifier value
    int token;                          // token type
    int reget_flag;          // character look ahead flag
    int keytype;                        // keyword type
    int line_count;
         char_t *cp;
         FILE * tp;                                          // temperory file to
 store procedure source code
         int sendsource;                           // flag to indicate source c
ode lines to be sent
    int newline;
public:
         token_t(char_t *c, FILE* t) { reget_flag = 0; line_count = 1; cp = c
;

         sendsource = 0; tp = t; newline = 0; };
         // forms tokens out of characters read from the source file
         int Next(void);
         long int Integer(void) { return(ivalue); };
         char *Identifier(void) { return(string); };
         int Type(void) { return(token); };
         void Reget(void) { reget_flag = 1; };
         int Keytype(void) { return(keytype); };
         int Line_num(void) { return(line_count); };
         void SendSource() { sendsource = 1; };
         void StopSend() { sendsource = 0; };
         int CheckSend() { return (sendsource); };
         char* Thisline() { return (cp->Thisline()); }
};


#endif
```

```
//------------------------------[ token.cpp ]---------------------------------
-

#include <string.h>
#include <iostream.h>
#include "..\head\char.h"
#include "..\head\token.h"
#include "..\head\errors.h"

extern error_t *ee;

/* Define Keywords */
static char *keywords[] = {
        "Ports",                                // KPORTS
        "End",                          // KEND
        "Input",                                // KINPUT
        "Output",                       // KOUTPUT
        "Devices",                      // KDEVICES
        "Coil",                         // KCOIL
        "Sensor",                       // KSENSOR
        "Pulse",                                // KPULSE
        "Programmable", // KPROGRAMMABLE
        "Plain",                                // KPLAIN
        "On",                                   // KON
        "Off",                          // KOFF
        "WaitOn",                       // KWAITON
        "WaitOff",                      // KWAITOFF
        "Send",                         // KSEND
        "Wait",                         // KWAIT
        "Strobe",                       // KSTROBE
        "Do",                                   // KDO
        "Procedure",            // KPROCEDURE
        "LPT1",                         // KLPT1
        "COM1",                         // COM1
        "COM2"                          // COM2
};

/* Scanner States */

#define SINI            0               /* Start (Initial) state */
#define SID             1               /* Identifier */
#define SINT            2               /* Integer */
#define STR                             3                       /* St
ring */
#define SQTE                            4                       /* Quote */
#define SLNF                            5                       /* Linefeed s
tate */
#define SDONE           10              /* final state */

short action_tbl[13][6] = {
            /*  States
                                        S   S   S   S   S   S
                                        I   I   I   T   Q   L
                                        N   D   N   R   T   N
                        I       T       E   F    */
/* CILL    */  0, 12, 14, 17, 19, 20,
/* CWHITE  */  1, 12, 14, 17, 19, 1,
/* CQUOTE  */  2, 12, 14, 18, 17, 20,
/* CID     */  3, 13, 14, 17, 19, 20,
/* CLPAREN */  4, 12, 14, 17, 19, 20,
```

```
/* CRPAREN */  5, 12, 14, 17, 19, 20,
/* CCOMMA  */  6, 12, 14, 17, 19, 20,
/* CDOT    */  7, 12, 14, 17, 19, 20,
/* CDIG    */  8, 13, 15, 17, 19, 20,
/* CCOLON  */  9, 12, 14, 17, 19, 20,
/* CSEMI   */ 10, 12, 14, 17, 19, 20,
/* CLF     */ 11, 12, 14, 20, 19, 11,
/* CEOF    */ 16, 12, 14, 20, 19, 16
};


/*  ********************************************************************** *
/

/*
   FUNCTION:    It scans the source file and classifies characters into
   different tokens

*/

/* ********************************************************************** *
/


int token_t::Next(void)
{
        // initialize the token state table
        int     state = SINI,
        slen = 0;

        // Reset reget flag if set and return token
   if (reget_flag)
        {
                reget_flag = 0;
                return(token);

#ifdef  VERBOSE
                if (token == TINTEGER) cout << ivalue << "\n";
                else cout << string << "\n";
#endif
        }
        do
   {
                cp->next();     // get next character to be parsed
                switch(action_tbl[cp->Class()][state])
                {
                case 0: /* illegal character */
                                token = TILLEGAL;
                                state = SDONE;
                                break;
                case 1: /* white space */
                                break;
                case 2: /* quote */
                                token = TQUOTE;
                                state = STR;
                                break;
                case 3: /* initial identifier character */
                                string[0] = cp->code();
                                slen = 1;
                                state = SID;
```

```
                                break;
                        case 4: /* left paren */
                                token = TLPAREN;
                                state = SDONE;
                                break;
                        case 5: /* right paren */
                                token = TRPAREN;
                                state = SDONE;
                                break;
                        case 6: /* Comma */
                                token = TCOMMA;
                                state = SDONE;
                                break;
                        case 7: /* Dot */
                                token = TDOT;
                                state = SDONE;
                                break;
                        case 8: /* Digit, initial */
                                ivalue = cp->code() - '0';
                                state = SINT;
                                break;
                        case 9: /* Colon */
                                token = TCOLON;
                                state = SDONE;
                                break;
                        case 10: /* Semicolon */
                                token = TSEMICOLON;
                                state = SDONE;
                                break;
                        case 11: /* Line Feed */
                                cp->Readline(); cp->reset_nextchar();
                                line_count++;
                                newline = 1;
//              state = SLNF;
                                break;
                        case 12: /* Identifier state/other character */
                                string[slen] = '\0';
                                cp->reget();
                                token = TIDENTIFIER;

                                // check for keyword
                                for (int i = 0; i < sizeof(keywords)/sizeof(k
eywords[0]); i++)
                                {
                                        if (strcmp(string,keywords[i]) == 0)
                                        {
                                                token = TKEYWORD;
                                                keytype = i;
                                        }
                                }
                                // Check for keyword so that keyword lines ar
e not printed into
            // the trace file
                                if ((token != TKEYWORD) && CheckSend() && (ne
wline))
                                        fprintf(tp, "%s", cp->Nextline());
            if (newline) newline = 0;
                                state = SDONE;
            break;
                        case 13: /* Identifier state/valid character */
```

```
                                if (slen < MAX_ID ) string[slen++] = cp->code
();
                                break;
                        case 14: /* Integer state/other character */
                                token = TINTEGER;
                                state = SDONE;
                                cp->reget();
                                break;
                        case 15: /* Integer state/valid character (digit) */
                                ivalue = (ivalue * 10) + (cp->code() - '0');
                                break;
                        case 16: /* End-of-file */
                                token = TEOF;
                                state = SDONE;
                                break;
                        case 17:  /* String initial */
                                string[slen++] = cp->code();
                                break;
                        case 18:  /* Quote encountered in String */
                                state = SQTE;
                                break;
                        case 19: // quote state other character
                                string[slen] = '\0';
                                cp->reget();
                                token = TCHARSTRING;
                                state = SDONE;
                                break;
                        case 20:
                                line_count++;
                                cp->reget();
                                newline = 1;
                                state = SLNF;
            break;
                        default:
                                printf("%s\n", ee->readerror(_F_PARSERERROR))
;
                                ee->checkerrors(FATAL_T);
                                break;
                }
         } while (state != SDONE);

#ifdef  VERBOSE
        if (token == TINTEGER) cout << ivalue << "\n";
        else cout << string << "\n";
#endif

        return(token);
}

/* end of file */
```

```
//--------------------------------[ trans.h ]---------------------------------
-

#ifndef TRANS_H
#define TRANS_H

#define SETBIT                  1
#define RESETBIT                0
#define MAXSTR                  80
#define MAXID                   30
#define EXTDELIMIT      '.'
#define BACKSLASH               '\\'
#define OUTEXT                  ".out"
#define REFEXT                  ".ref"
#define CMDEXT                  ".cmd"
#define TRACEXT         ".trc"


#include <stdio.h>
#include <string.h>

#include "..\head\char.h"
#include "..\head\token.h"
#include "..\head\stmts.h"
#include "..\head\list.h"
#include "..\head\crossref.h"
#include "..\head\errors.h"

extern list_t* procPtr;     // list class defined in list.h
extern char* strtrn(char* , const char);  // to be included in system library
extern error_t * ee;

// translator class
class translate_t {
    int stmt_ct;                //number of executable statements
    int error_ct;               // number of errors
    int warning_ct;     // number of warning errors
    token_t * t;
         char_t * c;
    char directory[MAXID];
         crossref_t cc;      // cross reference object
         char outfile[MAXID], refile[MAXID], tracefile[MAXID];
         FILE *pp;           // source file pointer
    FILE *trp;                  // temperory file pointer

public:
         translate_t(char* f)
         {
                        char *temp, *p;

                        stmt_ct=0;                                  // number of
lines compiled
                        error_ct=0;                                 // number of
compile errors
                        warning_ct=0;                       // number of warning
errors

                        if ((pp = fopen(f, "r")) == NULL)
         {
                                printf("%s\n", ee->readerror(_F_UNOPNSRCFIL)
```

```
);
                                    ee->checkerrors(FATAL_T);
        }
                        else
                        {       strcpy(directory, f);
                                p = strrchr(directory, BACKSLASH);  // retain
 only the directory
                                *(++p) = '\0';

                                temp = new char[strlen(f)+1];
                                strcpy(temp, f);

                                // strip off the source file extension
                                *(strrchr(temp,EXTDELIMIT)) = '\0';
                                strcat(strcpy(outfile, temp), OUTEXT);
                                strcat(strcpy(refile, temp), REFEXT);
            strcat(strcpy(tracefile, temp), TRACEXT);

#ifdef VERBOSE
                                printf("ouput file is %s\n", outfile);
                                printf("cross ref file is %s\n", refile);
#endif
                                c = new char_t(pp);
                                trp = fopen(tracefile, "w");
                                t = new token_t(c,trp);
                                procPtr = new list_t;

                        }
        }

        int parsePorts(void);
        int parseDevices(void);
    int parseProcedure(void);
        int generate(void);
        int crossrefer(void);
        FILE* Tempfile() { return (trp); }
    friend procedure_t::procedure_t(token_t* );
};

#endif
```

```
//------------------------------[trans.cpp]---------------------------------
-


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "..\head\functbl.h"
#include "..\head\trans.h"
#include "..\head\stmts.h"
#include "..\head\table.h"
#include "..\head\list.h"
#include "..\head\crossref.h"
#include "..\head\errors.h"

extern table* sytb;
extern list_t* procPtr;
extern tbltype functbl;
extern int numports;
extern error_t* ee;

// methods for translate object

/*  ************************************************************************ *
/

/*
   FUNCTION:    Begins to parse port declaration statements when the keyword
   PORTS is encountered. The parsing is continued by the port object when it
   is instantiated and inserted into the symbol table.

*/

/* ************************************************************************ *
/

int translate_t::parsePorts(void)
{
    int result = 0;

#ifdef VERBOSE
         printf("Parse Ports\n");
#endif

    result = ((t->Next() == TKEYWORD) &&  // check for keyword PORT
              (t->Keytype() == KPORTS));
    if (!result)   // if not generate error
         {
                printf("\n%s", t->Thisline());
                printf("Line %d %s\n", t->Line_num(), ee->readerror(_E_PORTKE
YEXPT));
                ee->checkerrors(ERROR_T);
         }
         else
         {
                // get all port declarations
      t->SendSource();
                while (!((t->Next() == TKEYWORD) && (t->Keytype() == KEND)))
                // make port entry into symbol table
```

```
                        sytb->insert(new port_t(t), "", KPORTS);
          }
          t->StopSend();
     return(result);
}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Begins to parse device declaration statements when the keywor
d
   DEVICE is encountered. The parsing is continued by the device object when
   it is instantiated and inserted into the symbol table.

*/

/* ********************************************************************** *
/

int translate_t::parseDevices(void)
{
     int result = 0;

#ifdef VERBOSE
          printf("Parse Devices\n");
#endif

     result = ((t->Next() == TKEYWORD) &&  // check for keyword DEVICES
               (t->Keytype() == KDEVICES));
     if (!result)     // if not generate error
          {
                printf("%s", t->Thisline());
                printf("Line %d %s\n", t->Line_num(), ee->readerror(_E_DEVKEY
WEXPT));
                ee->checkerrors(ERROR_T);
          }

          else
       // get all device declarations
                while (!((t->Next() == TKEYWORD) && (t->Keytype() == KEND)))
                        // make device entry into the symbol table
                        sytb->insert(new device_t(t), "", KDEVICES);
     return(result);

}

/*  ********************************************************************** *
/

/*
   FUNCTION:    Begins to parse procedure statements when the keyword
   PROCEDURE is encountered. The parsing is continued by the procedure object
   when it is instantiated and inserted into the symbol table.

*/

/* ********************************************************************** *
/
```

```
int translate_t::parseProcedure(void)
{
    int result = 0;

#ifdef VERBOSE
         printf("Parse Procedure\n");
#endif

        if (!feof(pp))
        {
                result = ((t->Next() == TKEYWORD) &&  // check for keyword PR
OCEDURE
                         (t->Keytype() == KPROCEDURE));
                if (!result)      // if not generate error
                {
                        printf("%s", t->Thisline());
                        printf("Line %d %s\n", t->Line_num(), ee->readerror(_
E_PROCKEYEXPT));
                        ee->checkerrors(ERROR_T);
                }

                else
                        // get all port declarations
      {
         t->SendSource();
                        while (!((t->Next() == TKEYWORD) && (t->Keytype() ==
KEND)))
                        // make port entry into symbol table
                                procPtr->append(new procedure_t(t));
      }
        }
        t->StopSend();
        fclose(trp);
        return(result);

}

/*  ********************************************************************** *
/

/*
    FUNCTION:   generates p-code for the functions in the procedure declarati
on.
    The p-code is output into a text file, which serves as input to the
    interpreter.

*/

/* ********************************************************************** *
/

int translate_t::generate(void)
{
   FILE* fp;
        FILE * rp, temp;
   char sendstr[MAXSTR], directionStr[MAXSTR];
   objptr_t code;
        long curpos;
   int len;
        char cmdfile[MAXID];
```

```
        symbol_t* ps;                    // to hold port symbols from the symb
ol table
        port_t* pt;                                     // port type pointer

#ifdef VERBOSE
     printf("Generate code \n");
#endif

        code = procPtr->first();

   // create output file with same filename as input file but with an
   // extension of 'out'
        if ((fp = fopen(outfile, "w")) != NULL)
        {
         if ((trp = fopen(tracefile, "r")) != NULL)
         {
                // generate pseudo opcodes for port declarations
                for (int i=0; i<numports; i++)
                {
                        fprintf(fp, "%d %s", CSOURCE, fgets(sendstr, MAXSTR,
trp));
                        // search for identifier in symbol table
                        if ((ps = sytb->getsymbol(i)) != NULL)
         {
                                switch (ps->type)
                                {
                                        case KPORTS:
                pt = sytb->convertPort(ps->symbol);
                                                switch (pt->get_type())
                                                {
                                                        case ADDRPORT:
                                                                switch (pt->g
et_direction())// Set up direciton string
                                                                {       case
KINPUT: strcpy(directionStr,"INPUT"); break;
                                                                        case
KOUTPUT: strcpy(directionStr,"OUTPUT"); break;
                                                                        defau
lt: break;
                                                                }
                                                                fprintf(fp, "
%d %u %s\n", CPORT, pt->get_portAddress(),
        directionStr);  // direction as a string
                        break;
                                                        case SERPORT:
                                                                fprintf(fp, "
%d %d %d %d %d\n", CSERPORT, pt->Baudrate(),
                                                                        pt->S
tartBit(), pt->StopBit(), pt->ParityBit());
                                                                break;
                                                        default:
                                                                break;
                                                }
                                                break;
                                        default:
                                                break;
            }
                        }
                }
```

```
     // generate pseudo opcodes for functions
     while ((code != NULL))
          {
                         fprintf(fp, "%d %s", CSOURCE, fgets(sendstr, MAXSTR,
trp));

                         switch (code->codes.opcode)
                         {
                                 case CON:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case COFF:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                         break;
                                 case CWAITON:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case CWAITOFF:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                         break;
                                 case CSEND:
                                         fprintf(fp, "%d %s %s\n", code->codes
.opcode, code->codes.comport, code->codes.string );
                                         break;
                                 case CWAIT:
                                         fprintf(fp, "%d %d\n", code->codes.op
code, code->codes.waitsecs);
                                         break;
                                 case CSTROBE:
                                         fprintf(fp, "%d %ld %d %d\n", code->c
odes.opcode, code->codes.address, code->codes.bit, SETBIT);
                                         break;
                                 case CDO:
                  // command files are stored in the user's directory
                                         strcpy(cmdfile, directory);
                 // append "cmd" extension for command files
                                         strcat(cmdfile, code->codes.filename)
;
                                         if ((rp = fopen(strcat(cmdfile, CMDEX
T), "r")) != NULL)
                                         {
                                                 while (!feof(rp))
                                                 {
                                                         if (fgets(sendstr, MA
XSTR, rp) != NULL)
                                                         fprintf(fp, "%d %s %s
", code->codes.opcode, code->codes.comport, sendstr );
                                                 }
                                                 fclose(rp);
                                         }
                                         else
                                         {       printf("%s %s\n", ee->readerr
or(_F_UNOPNCMDFIL), code->codes.filename);
                                                 ee->checkerrors(FATAL_T);
                  }
```

```
                                        break;
                                default:
                                        break;
                        }
                        code = (procedure_t*) procPtr->next();
                }
        }
        else
        {       printf("%s\n", ee->readerror(_W_UNOPNTRCFIL));
                ee->checkerrors(WARNING_T);
     }
     fclose (trp);
        fclose (fp);
      }
      else
      {       printf("%s\n", ee->readerror(_F_UNOPNOUTFIL));
              ee->checkerrors(FATAL_T);
              return(0);
      }
      fclose(pp);
      return (1);
}

/*  ********************************************************************** *
/

/*
        FUNCTION:   Builds and prints the cross reference table.  Refer to
    crossref::build for more information.

*/

/* ********************************************************************** *
/

int translate_t::crossrefer(void)
{
        typedef int ndxarr[10];

        ndxarr i1, i2;       // indices into the cross reference table

        cc.build(i1, i2);            // build cross references
        cc.print(refile, i1, i2);    // print into a separate file
        cc.display(refile);          // display cross references
        return (1);
}

/* end of file */
```

```
100 %_F_UNOPNSRCFIL Fatal error - Unable to open source file
101 %_F_UNOPNERRFIL Fatal error - Unable to open errors data
102 %_F_UNOPNOUTFIL Fatal error - Unable to create output file
103 %_F_EREADSRCFIL Fatal error - Error reading source file
104 %_F_PARSERERROR Parser error - undefined action code
105 %_F_UNOPNCMDFIL Fatal error - Unable to open command file
106 %_F_UNREAERRFIL Fatal error - Unable to read from error file
107 %_F_UNRESERRPTR Fatal error - Unable to reset error file pointer
111 %_E_SEMICOLEXPT Syntax error - Semicolon ; expected
112 %_E_FUNCOPREXPT Syntax error - funciton operator expected
113 %_E_UNDEFIDENTF Syntax error - Undefined identifier
114 %_E_IDENTIFEXPT Syntax error - Identifier expected
115 %_E_TYPADDREXPT Syntax error - Type name or address expected
116 %_E_INCORSERPOR Syntax error - Incorrect serial port
117 %_E_INTEGEREXPT Syntax error - Integer expected
118 %_E_INVALDEVTYP Syntax error - Invalid device type
119 %_E_DEVTYPEXPTD Syntax error - device type expected
120 %_E_INVALIDFUNC Syntax error - Invalid function name
121 %_E_KEYWORDEXPT Syntax error - keyword expected
122 %_E_PARTYPMISMA Syntax error - parameter type mismatch
123 %_E_PORTKEYEXPT Syntax error - expected keyword: Ports
124 %_E_DEVKEYWEXPT Syntax error - expected keyword: Devices
125 %_E_PROCKEYEXPT Syntax error - expected keyword: Procedure
150 %_W_UNOPNCRFILE Warning - Error opening cross reference file
151 %_W_BAUDNOTSPEC Warning - Baudrate not specified
152 %_W_STMTNOEFFEC Warning - Statement has no effect in code
153 %_W_UNOPNTRCFIL Warning - Unable to open trace file
```

```
* Port Declarations
Ports
        PortC 64259 Output;
        PortA 64256 Input;
     Com1port COM1 3600 3 2 1;
End

* Device Declarations
Devices
        PalletLiftUp Pulse PortC 4;
        Conveyor Coil PortC 5;
        PhotoCell Sensor PortA 7;
        PalletArrived Sensor    PortA   6;
        ChuckOpen Pulse PortC 1;
        Robot Programmable LPT1;
        Lathe Programmable Com1port;        * com port declared in port secti
on
        LatheStart Pulse PortC 2;
        LatheStop Sensor PortA 4;
        PalletLifted Sensor PortA 5;
        PalletStops Coil PortC 0;
        ChuckClose Pulse PortC 3;
        PalletLiftDown Pulse PortC 6;
        LatheRunning Sensor PortA 2;
        LatheHandShk Sensor PortA 3;
        Delay Wait;
End

* Procedure Declarations
Procedure
        Lathe.Do(loadlathe);
        Robot.Send("NT");
        PalletStops.On;
        Conveyor.On;
        PhotoCell.WaitOn;
        PalletStops.Off;
        PalletArrived.WaitOn;           * Wait on part arrival
        Delay.1000;
        PalletLiftUp.Strobe;
        PalletLifted.WaitOn;
        Conveyor.Off;
        ChuckOpen.Strobe;
        Robot.Do(loadpart);
        Delay.1000;
        ChuckClose.Strobe;
        Delay.2000;
        Robot.Do(moveaway);
        Delay.2000;
        LatheStart.Strobe;
        LatheStop.WaitOff;
        Robot.Do(MoveBack);
        Delay.2000;
        ChuckOpen.Strobe;
        Delay.2000;
        Robot.Do(GetPart);
        PalletStops.On;
        PalletLiftDown.Strobe;
        Conveyor.On;
        Delay.500;
        Conveyor.Off;
```

```
        LatheStart.Strobe;
        PalletStops.Off;
End
```

```
* Port Declarations
Ports
        PortA 64256 Input;
        PortC 64257 Output;
        Com1port COM1 3600 3 2 1
End

* Device Declarations
Devices
        PalletLiftUp Pulse PortC 4;     * Pulse device
        Conveyor Coil PortC 5;
        PhotoCell Sensor PortA 7;
        Lathe Programmable Com1port;
        Robot Programmable LPT1;
End

* Procedure Declarations
Procedure
        Robot.Send("NO");
        PalletLiftUp.Strobe;

        * Turn the conveyor on
        Conveyor.On;

        PhotoCell.WaitOn;
        Conveyor.Off;
        PalletLiftUp.Strobe;
   Lathe.Send("NT");
End
```

APPENDIX H: CPL Language Reference & User Manual

1. Introduction

The Cell Programming Language (CPL) is a high-level special purpose language being developed at the Department of Systems Analysis at Miami University. This project is part of a larger project to design a computer aided manufacturing system, and support course work, projects, and research in Flexible Manufacturing.

1.1. What is CPL

CPL is a programming language environment for use in the control of manufacturing cells. Individual cell components and their operations can be integrated by programming the cell as a single unit. Programs to do this could be written in any other existing high-level language such as BASIC or C, but the user would have to be familiar with the syntax necessary to perform low-level input and output to the various hardware devices that provide the interface to the cell's devices. For example, the user would set a particular bit on a particular hardware port to 1 to turn on a device. Instead, CPL allows the user to program the cell by using commands such as On and Off, and the CPL system will take care of the low-level programming details.

CPL does not hide all of the hardware details. In order to use CPL, the user is still required to know the particular hardware device and bit to which each device is interfaced. Also, the user must know the type of device. Finally, individual cell components such as robots and CNC machines will have to be programmed in their host languages. One advantage, however, is that the programmer has full control of the operations, and can communicate with the individual devices even after the programs have been loaded into their memories.

1.2. The CPL Environment

The CPL environment consists of :

1. CAD/CAM workstations, a file server and peripherals;
2. A local area network;
3. Personal Computer (PC) controlled manufacturing cells;
4. Interfacing electronics between the PCs and the cell devices;
5. A programming language used to program the cells.

An overview of the environment can be found in the paper "Object-Oriented Flexible Manufacturing System at Miami University" in Appendix A of this document. The remainder of this document is devoted to the description of item 5 in the above list.

2. How CPL Works?

The CPL software consists of three major components:

1. CPL compiler;

2. CPL interpreter;
3. Remote status display.

The CPL compiler processes the user's CPL program along with any required robot and/or CNC command files to produce an intermediate file of instructions known as p-code. This p-code is the input to the interpreter which performs the low-level input/output operations on the cell controller PC. Thus, the compiler can be run on any PC or CAD/CAM workstation, but the interpreter must reside on the cell controlling computer. Once a CPL project has been compiled to p-code, it need not be recompiled unless a change is made in the CPL code. The machine command files serve only as input so any changes made to them will not affect the execution of the CPL program. A debugging option is provided in the interpreter which helps to eliminate CPL program errors.

The remote status display is an optional component of the system that can be used to remotely monitor the operation of the cell. The remote status display has a component, called the monitor, that runs on the cell controller, and a component, called the display, that runs on a remote PC, for example a CAD/CAM workstation. The monitor sends the state of each device to the display which in turn outputs the status to the user.

3. A CPL Project

Earlier it was stated that CPL is a language that allows the user to control and integrate devices of a manufacturing cell, but that the user is still required to program individual cells in their host languages. Thus, a CPL program would consist of:

1. A CPL program, and
2. Zero or more command files for programmable devices.

In managing a project, the user should keep all of the command files and the associated CPL language file together, ideally in a separate directory on a CAD/CAM workstation.

4. A CPL Program

A CPL program consists of five major sections: port declarations, device declarations, cell declarations, procedure declarations, and program declarations. The following subsections elaborate on each of these.

4.1. Objects and Classes in CPL

A CPL program has three major types

1. Ports: Used to name hardware interface ports.
2. Devices: Used to name individual cell devices, and assign ports and bit numbers
3. Cells: Used to declare network addresses of cell controlling computer.

In CPL, all data types are classes and the instantiation of an object to be of a type or class

assigns values to the attributes of the of the object. However, a dynamic change to the value of an attribute is not possible.

4.1.1. Port Declarations

The port declaration section is used to assign a physical port address on the PC. The declarations are made within a PORTS.... END block. Following the keyword PORTS is a series of individual port declarations. The syntax of a port declaration is as follows

< port_variable > (< port_address > < direction >)|(< port_name > < baudrate > < data_bits > < stop_bits > < parity >);

The *port_variable* can be any user defined identifier consisting of a maximum of 31 characters. The identifier can consist of alphabetic characters, digits and underscores upto a maximum of 31 characters. The *port_address* should be a physical port address and the *direction* is either INPUT or OUTPUT depending on whether the port is used to send or receive signals; the default direction is INPUT. If the port is a serial port, the *port_name* should be one of the serial ports COM1: or COM2: followed by the *baudrate*, number of *data bits*, number of *stop bits*, and the type of *parity* should be specified. An example port declaration section is given below.

```
Ports
        PortA  64259  Output;
        PortB  64256  Input;
        PortC  64257;
        Com1port COM1: 3600 7 1 1;
End
```

4.1.2. Device Declarations

The device declaration section is used to declare a device object and associate a port and bit number with it. The device types are predefined and correspond to the devices in the cell. We have not made provisions for including user-defined device types in the language, because at this juncture we do not anticipate such a need. The declaration block is bounded by the keywords DEVICES and END. The syntax for the device declaration is as follows,

< device_variable > < device_type > (< port_variable > [< bit_number >]) | < programmable_port > ;

The *device_variable* is a user defined identifier and the *device_type* is a keyword in the language. The *port_variable* should have been defined earlier in the port declaration section, and the *bit_number* is a constant between 0 and 7 and corresponds to a bit on the data acquisition board. For a programmable device type, the port name LPT1 is specified if a parallel port is to be used, and the port identifier that has been assigned one of the serial ports COM1: or COM2: is specified if a serial port is to be used. An example device declaration section is given below.

```
Devices
 PalletLiftup    Pulse    PortC  4;
```

```
Conveyor    Coil    PortC 5;
Robot       Programmable LPT1;
Lathe       Programmable Com1port;
End
```

4.1.3. Cell Declarations

The cell declaration section is used to assign network addresses to cell names in order to provide for communication between cells. Declaring cell names makes it convenient to assign a procedure to a cell, and facilitates modular programming at a small scale. The syntax of a cell declaration is as follows:

< cell_variable > < network_address >;

The *cell_variable* is like any other user defined identifier, and cannot exceed a maximum of 31 characters. The *network_address* is a pre-defined host name or network address that has been assigned to the cell controlling computer by the system administrator. An example of a cell declaration section is given below.

```
Cells
        ManufacCell     cimlab6;
        StorageCell     192.34.54.3;
End
```

4.2. Control Constructs in CPL

A CPL program has two basic control constructs:

7. Procedures: Contains the sequence of cell control operations executed on devices
8. Program: Collection of procedures to be executed on cells

4.2.1. Procedure Declarations

The next section in the program is the procedure section which consists of statement constructs. Each statement represents one device operation and directly corresponds to an actual operation of the real device. The syntax of a procedure statement is as follows,

< device_variable > . (< device_function > [< open_parenthesis > parameter { , . . . }. < close_parenthesis >]) |
< delay_time >

The *device_variable* is an identifier previously declared in the device declaration section. The *device_function* is predefined, and is a keyword in the language. Table 4.2.1 lists device types and valid functions for each device type. Function parameters are enclosed within parenthesis and are separated by commas. As with devices and ports, the keywords PROCEDURE and END mark the beginning and end of a procedure block. An example of the procedure section is given below.

```
Procedure
        Conveyor.On;
        Robot.Send("NT");
```

```
        Lathe.Do(Cutpart);
        Delay.1000;
End
```

TABLE 2.1

TYPES	VALID FUNCTIONS
COIL	ON, OFF
SENSOR	WAITON, WAITOFF
PULSE	STROBE
PROGRAMMABLE	SEND, DO
DELAY	MILLISECONDS

Students can program device objects with names that directly correspond to their real-world counterparts. The predefined device functions are named after the actual device operations. For e.g., a statement such as Conveyor.On is an instruction to switch on the conveyor and a statement such as PhotoCell.WaitOn is an instruction to wait for the photocell to be switched on. This way it is possible to write a program and visualize an entire production operation without actually performing it.

The user is, however, required to program a programmable device type using its host language. Commands to the PROGRAMMABLE device type can be given directly by passing them as parameters to a function or they can be stored in a separate file, and the file name passed as the parameter. For e.g., Robot is declared to be a programmable device, and the Send operation accepts a parameter which is a string and sends a command to the robot. The Do function on the other hand accepts an identifier that is the name of a file consisting of robot commands which are read and directly output to the robot.

4.2.2. Program Declaration

The program section is the last section in a CPL program, and is simply a series of statements arranged in a predetermined order by the programmer. Each procedure statement states what procedure is going to be executed on what cell, and specifies repetition clauses, if any. The program statements appear within a PROGRAM....END block. The syntax of a program statement is as follows:

< cell_name >. < procedure_name > [(< condition > | < repetition_times >)];

The *procedure_name* is the name of a procedure that has been defined previously. Similarly the *cell_name* is also the name of a cell that has already been declared. Repetition clauses, if any, must be specified either as a *condition,* or as the number of *repetition_times* that a proce-

dure is to be executed. These are enclosed within parentheses. An example program declaration is given below.

```
Program
        ManufacCell.ProduceBody;
        StorageCell.StoreParts (ManufacCell.SignalOn);
        ManufacCell.ProduceBody(50);  * Repeat 50 times
        StorageCell.StoreParts(50);
End
```

In CPL, every statement excepting block markers end with a semicolon. The programmer may insert comments in the program by preceding the comment with an asterisk, which is the comment character. A valid program statement and a comment may be typed on the same line, but the comment should follow the program statement. The reverse is, however, not true, because all characters in a line following a comment character are ignored by the CPL compiler. The syntax of this language is kept simple and compact in order to make it more appealing to users. At the end of the compilation, a cross-reference output listing the cross referencing between various devices and ports is printed. Error handling is performed by an error object which prints out error messages with corresponding line numbers and error codes. An example of a complete CPL program is given in appendix E.

5. Using CPL

To invoke the CPL system from the network, type the command

CPL *drive pathname filename*

The *drive* is the drive on which the input and program files are located, and the *pathname* is any absolute or relative DOS path specification. Absolute path specifications are given from the root directory and relative path names are given from the current directory. The *filename* is any combination of alphabetic characters, and the entire file specification should not exceed thirty characters.

The CPL program file can be edited using any standard editor, and should be stored as an ASCII text file. One advantage of structuring the CPL system as two independent components is that the execution is not tied up with the compilation process. To execute the CPL, program type the command

CIMINT *drive:pathname filename.*

The CPL program can also be executed in the debug mode. To do this, type the command

CIMINT –d *drive pathname filename.*

The debug mode will execute the program in steps and the user can trace through the program. This is especially useful for locating program errors.

6. CPL Errors

Errors in the program are detected by the compiler and displayed onto the screen at the end of the compilation. At this point errors are not output into a list file, but this feature may appear in future versions of the compiler. CPL compiler errors are of three types: Syntax Errors, Fatal Errors, and Warning Errors. Currently the compiler allows a maximum of five syntax errors before it terminates abnormally. Fatal errors are those that make it impossible for the compilation process to continue. As such the compiler terminates execution immediately after a fatal error is discovered; no count of fatal errors is kept. Warning errors are those that do not seriously affect the production of p-code, but may produce unpredictable results at run-time, or impede the debugging process by producing incorrect cross references. Although warning errors are detected and displayed, they are ignored by the compiler. A description of the standard error messages produced by the CPL compiler are listed in the following pages, along with hints to rectify the errors.

6.1. FATAL Errors

UNOPNSRCFIL - Unable to open source file

The compiler was unable to open the source file, because of incorrect path specification or access violation. Restart compiler with correct path name, and check access protection for the file.

UNOPNERRFIL - Unable to open errors database

The compiler was unable to open the file containing the compiler error messages. Consult the system administrator or the person in charge of maintaining the CPL system.

UNOPNOUTFIL - Unable to create output file

The compiler was unable to create the output p-code file. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name.

EREADSRCFIL - Unable to read source file

The source program file contained some extraneous characters which the program could not decipher, or the compiler does not have read access for the file. Check the source file for any extraneous characters, and also check the group and world access protection for the file.

PARSERERROR - Undefined action code

A bug in the CPL compiler. Consult the system administrator.

UNOPNCMDFIL – Unable to open the command file

The compiler was unable to open a command file specified in the program. Check the name of the command file specified for spelling errors, and also check if the command file is present in the same directory as the input file.

UNREAERRFIL – Unable to read from errors file

The compiler was unable to read the file containing the error messages. Consult the system administrator.

UNRESERRPTR – Unable to reset error file pointer

The compiler is unable to reset the file pointer for the error file. Consult the system administrator.

6.2. SYNTAX Errors

SEMICOLEXPT – Semicolon ; expected

A semicolon was not found where expected. Insert a semicolon at the end of the statement appearing on the line indicated by the line number in the error message.

FUNCOPREXPT – Function operator expected

The function operator '.' was omitted in a procedure statement. Insert the function operator after the device name in the procedure statement appearing on the line indicated by the line number in the error message.

UNDEFIDENTF – Undefined identifier

The identifier discovered was not declared previously in a declaration section. Enter a declaration for the identifier in the corresponding declaration section.

IDENTIFEXPT – Identifier expected

The compiler was expecting to find an identifier, but could not find one. Include the necessary identifier in the statement appearing on the line indicated by the line number in the error message.

TYPADDREXPT – Type name or address expected

A port variable was not assigned an address or a serial port type. Insert a port address or serial port name in the statement appearing on the line indicated by the line number in the error message.

INCORRSERPORT – An incorrect serial port was specified

An incorrect serial port was specified in the declaration of a serial port variable. Check the serial port name in the definition for predefined serial ports, and correct spelling errors, if any.

INTEGEREXPT – Integer expected

An integer was not found where expected. Check program to locate actual error and make the necessary changes.

INVALDEVTYP – Invalid device type

The device type specified in a device declaration is invalid. Check declaration for spelling errors, and change it to a known device type. If error persists even after correction, consult the system administrator.

DEVTYPEXPT – Device type expected

The compiler was expecting a device type, but did not find one. Check the syntax and make necessary changes.

INVALIDFUNC – Invalid function name

The function name specified is not valid for the accompanying device type. Check the manual for permitted functions for the device type specified, and make the necessary changes.

KEYWORDEXPT – Keyword expected

An was not found where expected. Check program to locate actual error and make the necessary changes. If error persists even after correction, consult the system administrator.

PARITYMISMA – Parameters type mismatch

The parameter type specified does not match with the type expected.

PORTKEYEXPT – expected keywords: Ports

The port declaration section did not begin with the keyword, Ports.

DEVKEYWEXPT – expected keywords: Devices

The device declaration section did not begin with the keyword, Devices.

PROCKEYEXPT – expected keywords: Ports

The port declaration section did not begin with the keyword, Procedure.

6.3. WARNING ERRORS

UNOPNCRFILE – Unable to open cross reference file

The compiler was unable to create or open the cross-reference file. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name. This error does not stop the generation of p-code in the output file, but does not produce useful debugging information.

BAUDNOTSPEC – Baud rate not specified

The baud rate for the serial port specification was not specified in the port declaration section. The compiler will assume the default baud rate which may not match the actual baud rate for the connected device.

STMTNOEFFEC – Statement has no effect in code

A procedure statement or program statement was discovered without any function name. Such a statement is meaningless, and does not produce any p–code.

UNOPNTRCFIL – Unable to open trace file

The compiler was unable to create or open the trace file, which is a temporary file used to output source code statements needed to send trace information to the interpreter. Check the drive status if drive included in the path specification. Restart the compiler with the correct path name. This error does not stop the generation of p–code in the output file, but does not produce useful debugging information.