Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1992

Software Development for Manufacturing Systems- Language and Networking Issues

Shabi Farooq
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1992-013

Software Development for Manufacturing Systems- Language And Networking Issues

Shabi Farooq

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

Software Development for Manufacturing Systems - Language and Networking Issues

by

Shabi Farooq
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #92-013 10/92

Software Development for Manufacturing Systems - Language and Networking Issues

by Shabi Farooq

Department of Systems Analysis

Miami University

Oxford, Ohio

Submitted in Partial Fulfillment of the Requirements of the Masters Degree in Systems Analysis

October 15, 1992

Abstract

We have witnessed unprecedented changes in the industrial world with the advent of computers and the field of manufacturing is no exception. With the boom of microcomputers, their usage in manufacturing systems was realized at every level – from the shop floor level to the administrative and management layers. This paper deals with the software development issues that a software engineer has to take into account when analyzing, designing, and implementing software for manufacturing systems. Two important criteria that one has to consider are the real–time requirements and the device independent abstractions that such software has to provide to the end–user, since it is reasonable to expect an end–user to know very little about the software intricacies. Two specific aspects of manufacturing software are discussed in detail here. The first part discusses a language with an object oriented flavor for programming manufacturing systems. In particular, some of the design aspects and implementation issues are discussed. The other part describes the networking issues that are specific to the manufacturing environment. A prototype manufacturing system developed as a part of the project is used as a model to explain the various concepts and issues. A detailed description of the new language, ***Cell Programming Language*** *(CPL), developed for the prototype is also included.*

1.Introduction

Manufacturing, which dates backs to the Stone Age, has evolved from an ad hoc and imprecise technique to a very sophisticated and mathematically sound engineering discipline. The reasons for this are twofold : 1. increasing need to produce goods which are of high quality, 2. need to optimize cost, effort, and time. These criteria point unanimously in one direction – Automation. In addition to eliminating the human component from repetitive task execution, automation also ensures *repeatability* which is very crucial to promoting quality and maintaining it in a consistent manner. It also helps in enforcing high standards of quality by rigorous specification of the requirements which can then be built into automated tools. In the past two decades or so, the approaches to automation itself have undergone tremendous changes. Simple automated manufacturing systems like *copy lathes* gave way to machines with hard–wired logic circuits, which took input in form of simple instructions and executed them faithfully and consistently. These machines which came into being in the early and mid–70's, used to employ a *programmable*

controller and are commonly referred to as *numerically controlled* machines for their ability to read in commands in the form of numbers and other symbolic representations. The significant feature, which was clearly a drawback, was the fact that most of the logic and control had to be *hard–wired* into the controller unit, thus leaving very little room for flexibility in control and management of the machines. This realization resulted in a totally new approach, involving computer–based control, commonly known as *Computerized Numerical Control (CNC)* of manufacturing systems. Even though computers were in use prior to this period, their prohibitive costs made it almost impossible to incorporate them into the manufacturing environment. It was the boom of mini and micro computers which triggered this revolution and made it possible to overcome the drawbacks associated with numerically controlled (NC) systems. The advantages realized by such an approach are [1]:

1. An increase in flexibility,
2. A reduction in the complexity of the hardware circuits, as well as the availability of automatic diagnostics programs, brings a subsequent need for fewer maintenance personnel,
3. A reduction in inaccuracies in manufacturing due to a reduced use of the tape reader,
4. An improvement in the possibilities for correcting errors in part programs – the *editing* feature,
5. The possibility of using the computer's peripheral equipment for debugging the edited part program; e.g., a plotter can be utilized for drawing the shape of the part.

The other significant aspect of the usage of computer–based control was the ability to establish reliable communication channels between various manufacturing entities through computers which could be networked together. In fact, this approach could be extended to include the design and management components of manufacturing systems to create integrated design and manufacturing environments also known as *Computer Aided Design and Manufacturing Systems* (CAD/CAM).

With this overview about the manufacturing world today, it is now time to look at the implications of such a revolution on the software, hardware, and networking requirements from the point of view of

software engineering and computer science, which is the domain that our work lies in. The foremost realization is the stringent requirement on the timely execution of commands and instructions which clearly forces the software and the hardware into the *real time domain*. A specific instance of such a software design which highlights real time factors is the development of a software based robot controller [2]. In addition, the life cycle for such software is more task oriented as opposed to being data oriented [3].

In the network sub-component, time constraints become extremely crucial when process control has to be performed remotely, i.e., from a computer that is not directly wired into the manufacturing cell. This not only puts a demand on the software development at the application level, but also poses strict real time requirements at the low-level software managing the network.

Another feature inherent to the manufacturing environment is the lack of a common programming syntax for programmable manufacturing standards. Lack of such standards makes programming such devices in their native programming syntax very cumbersome and annoying. In addition, the primitive nature of instructions, which comprise a set of primitive opcodes and operands, does very little to alleviate the burden of programming.

The aforementioned aspects form the central core of the project which are discussed in detail in the following sections.

1.1.Description of the Problem and Motivation for the Project

The Computer Integrated Manufacturing Lab at Miami University, which is used for teaching undergraduate lab courses in the Manufacturing Engineering curriculum, lacked an environment that could group the various disparate units like the machining centers, conveyors, robots, and material storage and retrieval systems into independent programmable flexible manufacturing *cells*. In other words, there was no common software development platform on which students could easily program these various machines in a high level language, thus making it unnecessary to know the specific commands for each of the machines. Further, the existing engineering design software had to be incorporated into this platform so that a prototype *Computer Integrated Manufacturing* (and design) environment (CIM) could be developed. This was the practical motivation. The theoretical motivation was to test the object-oriented paradigm to develop software for a such a system and to test the suitability of the same to real-time application development in general. We also believed that this approach could easily accommodate changes

in the system when new components were added or existing specification of the machines were changed [4].

With this in mind, we set out to define the problem domain for which we could develop a solution, which in turn, could be extrapolated to accommodate further changes and enhancements in the problem domain. The domain thus comprised the following physical entities:

6. A CNC machining center with a pneumatically controlled chuck,
7. A spatial robot with three degrees of freedom which accepted commands from a programmable linear controller,
8. A conveyor system,
9. Various electro-pneumatic actuators and feedback devices like the photo sensitive and limit switches,
10. A set of stand alone personal computers,
11. Some graphics and engineering design software running on the PCs mentioned above, and
12. A RISC machine which could act as a server for a Local Area Network (LAN).

1.2.Outline of the Paper

This project was pursued by two graduate students working along with three faculty members. Two distinct phases were recognized for the given task. The first one dealt with the language semantics and design issues and the second one dealt with networking issues. Other aspects that were recognized along the way are discussed in detail in the section on future directions.

Section two discusses the salient features of the o-o paradigm and in particular its suitability to manufacturing systems software design as well as its drawbacks. Issues like device independent software and maintenance are also discussed. The remainder of the section deals with the real-time issues pertaining to the manufacturing environment which in turn expose the limitations of this approach.

In the third section, description and implementation aspects of the CPL and the associate translation tools are discussed. The development of these tools was also done in an object-oriented manner. Im-

plementation of real-time constraints, though modest in number, are also explained. Limitations of the system in the real-time domain are also be outlined in the section dealing with summary and conclusions.

The fourth section discusses the network design including LAN topology and the protocols adopted for communication between the computers. The network-based tools developed for remote monitoring, as well as data communication functions of the network are discussed in detail.

This paper concludes with a summary of our experiences and an outline of the limitations of the present configuration. Following this, future directions are discussed as an aid to other students who may be interested in pursuing this project further.

2.Object-Oriented Methodology and Manufacturing Systems

The important requirements in designing software fo computer-controlled manufacturing systems are :

13. To provide a common platform across various manufacturing peripherals and
14. To hide the physical details of these peripherals from the end user as far as possible and at the same time, make the software independent of the actual peripherals that it represents or operates.

It is interesting to look at the second requirement in detail. To remove the physical details of the device from the user's view means that the language should specify generic actions. Consider an example: *start lathe*. This command does not specify how to start the lathe, but just specifies an action to be performed on the lathe. It does not specify whether a solenoid switch has to be tripped or a hydraulic switch has to be turned on. Thus, by providing such an instruction, we have created an *abstraction* of the act of switching on a device. The latter part of the requirement seems to contradict the first one in that, once we provide a implementation-independent abstraction to the end-user, should we not handle the implementation-specifics within the software. This apparent problem can be very easily circumvented by moving those details into the hardware of the computer that controls the switches and actuators of the various manufacturing paraphernalia. To decide what the hardware should handle and what the software should is a non-trivial issue in the design of real-time system development [5].

The need for a common platform and device-independent software calls for an approach which can map entities into the problem domain to the software while hiding the actual representational details of the same. Also, we desire to have a platform that can span different components of manufacturing thus providing a truly integrated environment [9]. Assuming that this can be done, the next choice is to decide which of the three approaches - process-driven, data-driven, object-oriented, or a mix of the two. Process-driven approach, although sufficient, does not do a good job of providing a good abstraction of the system on the software level to the user. Data-driven approach is not suitable at all since data do not form the core of the system. Real-time command execution and data-acquisition form the central theme of these systems. The o-o paradigm fits into this situation very well for three main reasons:

1. It provides a very nice abstraction (by virtue of class concept and data encapsulation) of the manufacturing peripheral it represents by incorporating the *data* (that the peripheral would manipulate) and *actions* (that it can perform for the external user) into a *class,*
2. Additions or modifications made to the existing set up can be very easily incorporated into the software with practically no change to user-level abstraction, as long as the access interface remains unchanged, and
3. A homogeneous interface can be built to encompass all the sub-components of manufacturing with such an approach.

These reasons are discussed in more detail in the next section. Justification for this approach is also discussed in [9].

2.1.Application of the O-O Approach to Design of Software for Manufacturing Systems

It is important to ask ourselves whether all manufacturing is object-oriented. This question is well addressed by the table below [6].

Manufacturing Objectives	Object-Oriented Premises
Production of concrete, well defined, repeatable, interchangeable parts and products.	Focuses all objects and their attributes, applying classification structures, encapsulation, and uniform representation.
Manufacturing by process and assembly plans	Applies assembly structures and high-level abstraction

Based on methods to define process operations and manufacturing services.	Methods define operations and services by objects.
Complex product, process and facility are designed by their elements.	Specifications of complex systems by their elementary object components.
Combines information and material processing to create end-products.	The O-O model combines the data and process model
Depends on systematic and consistent plans.	Consistent, systematic representation of reality.
Group technology/commonalty for productivity gains.	Delegation and inheritance by explicit representation of commonalty.
Communication and integration among multiple enterprise functions.	Communication by messages, polymorphism, and corresponding operations.
Culture of clear, ordered, and stable processes and procedures.	Model stability, clarity, and flexibility by minimal dependency between objects.
Performance: Maximum profit; minimum cost; rapid adaptation to change; quality.	Allows changes with minimum rewrite of code, minimum errors, improved management of complexity, and increased programmer productivity.

Table 2.1.1: The affinity between manufacturing and object-orientation

It should be noted here that the above table draws parallels of which only a few are relevant to this project. Nevertheless, this table does imply that o-o paradigm can be successfully used to develop an integrated manufacturing software environment that effectively encompasses the various components of a manufacturing system. This is pictorially depicted in the figure below[6].

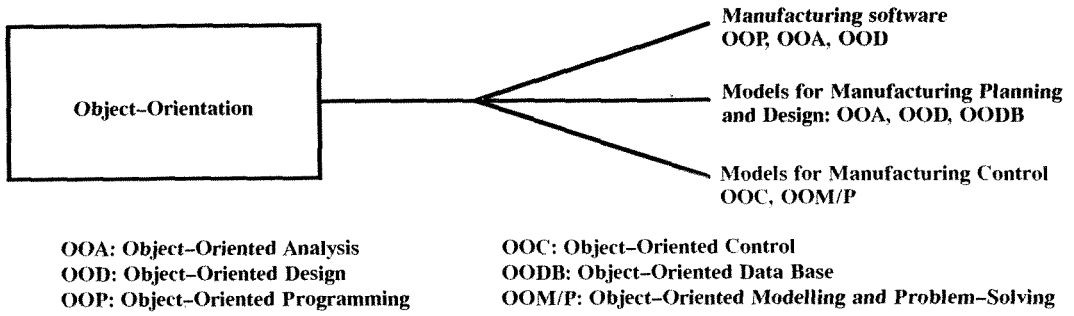


Figure 2.1.1: Scope of object-oriented manufacturing

These illustrations strongly indicate that manufacturing can be very closely modelled using the O–O approach. Additional material on this topic can be found in [7], [8], and [9]. The advantages of such an approach are summarized below [6]:

1. The entities in the abstract domain (i.e, class objects) have a natural one–to–one corresspondence with the entities in the physical world,

2. Modularity is inherent to this approach which has a very significant implication. With very little programming effort, the control software can be easily adapted to different environments,

3. Software development task itself can be simplified by dividing the effort among team members. Specific tasks need to be allocated by just specifying the abstract objects, leaving the implementation details to the particular member developing the software.

Unfortunately, this approach does not come without drawbacks. Recalling the real time constraints that have been outlined in the previous sections, some of the features of the o–o paradigm that make it very powerful turn out to be serious bottlenecks during implementation of o–o based software for manufacturing systems [10]. In the aforementioned reference, the author gives a detailed explanation about the drawbacks of such an approach. Since manufacturing systems are inherently distributed, remote object invocation feature is necessary which the present implementations of the paradigm do not support [6]. These features need to be built into the system using networking toolkits which in turn adds more time overhead to the over all real–time response. Nevertheless, it is important to note that the overhead of distributed support results in additional overhead due to message passing, which can deteriorate the performance further.

Another issue that is of importance in such an environment is the need for *shared memory*. In the manufacturing environment, it is common for two software processes to communicate with each other via a common memory area in the computer. In order to ensure consistency of information in such memory locations, it is important that at most one process access it at anytime. To achieve this the lan-

guage needs to provide features like *semaphores* and at this point no implementation of the o-o paradigm supports such a construct.

The other disadvantage that directly stems from the "power" of o-o paradigm is the overhead associated with *polymorphism* and *dynamic binding.* These features require the run-time environment o determine the specific method that needs to be invoked rather than determining it at compile time. This results in very high run-time costs that can seriously impede the real-time constraints. This is of crucial importance to *hard real time systems* since it can produce incorrect results in such systems. *Soft real time systems*[1] are less susceptible to such phenomena but, nevertheless, output could still be degraded [5]. Languages supporting those features also lack built-in support for process synchronization [6].

The drawbacks that we mentioned earlier can be overcome in part by having efficient hardware. In fact, we all hope that the emerging technology and future capabilities will solve this problem or at least provide a way to compensate for it.

3.Design and Development Cell Programming Language (CPL) and Associated Tools

It may be recalled from the requirements of a language for manufacturing environment that it has to provide a good abstraction of the physical devices and their activities that it represents and also be device independent. The language presented here, despite being independent of the specific details of the physical entities that it represents, is to a certain degree dependent on the hardware that provides the interface to the manufacturing devices. This still maintains the device independence since the exact details of manipulating the different peripherals is within the hardware and external electro-mechanical interfacing equipment.

On the other hand, the user now has to configure the hardware interfaces through software by using specific configuration commands that the language provides. These will be explained in more detail in the next section. It is important know that this dependency is inherent and hence unavoidable. One possibility to hide these details would be to hardcode these details into the translation tools for CPL, but this would introduce a certain amount of inflexibility and in turn make the language device dependent. So it has to be remembered that we are maintaining a delicate balance between device independent abstraction at the user level and the device-independent nature of the language itself.

1. Hard real time systems are those that have very stringent response requirements. Soft real time systems, on the other hand, can handle some amount of delay.

3.1.A Formal Description of Cell Programming Language

The formal specification of the grammar of CPL is included in appendix A. A CPL program consists of four major sections:

1. Port Declarations: Used to identify hardware interface ports.
2. Devices Declarations: To identify different manufacturing peripherals within a cell, and to assign ports and associate bits with these devices
3. Cell Declarations: Used to declare the network address of the cell controlling computers.
4. Procedure statements: Commands to execute manufacturing, monitoring, and feedback instructions.

3.1.1.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 key word *Ports* is a series of individual port declarations. Its syntax is as shown below:

```
<Port_Identifier> (<Port_address> <direction>) | (<baudrate> <data_bits>
                                                  <stop_bits> <parity> );
```

The *Port_Identifier* can be any alpha-numeric character string to identify a port with a user-defined name. Underscores may be used for forming descriptive identifiers. The *port_address* should be a valid hardware address that corresponds to an actual physical port in the computer hardware. It is this port that actually provides the interface to the physical domain, i.e., various manufacturing components. The direction indicates whether the port is used to input information or output information. Output drives some physical devices and input gets feedback or status information that is used to decide the course of future action. The default mode is input.

If the other specification is used, it implies the identification of a *serial communication port* that is usually used to downloading information to the external devices, i.e., sending commands to a lathe in its primitive instruction format or download a file consisting of similar instructions. It should be noted that these commands can be separately generated by other CAD/CAM software and then stored in files which can be accessed by the CPL programs written by the end-user. Thus the end-user can use these files without having to deal with the low-level instructions directly and thus maintaining the abstraction outlined in the earlier sections. Another type of port that is used in this set-up is the *printer port* which

can be implicitly addressed by the individual instructions. The reason for the lack of formal specification of such a port is simply because there is no formal setting of parameters to be performed when dealing with them. An sample piece of code to illustrate the use of these instructions is given below.

```
Ports
        PortA       64259       Output;
        PortB       64256       Input;
        PortC       64257;
        CommPort1   3600 7 1 1;
End
```

3.1.2.Device Declarations

In this section, each device in the cell is associated with a *bit* on a *port*. These devices have to be one of the *device_type* types which are defined as a part of the data type subset of the language. These predefined types correspond to physical entities in the cell. Any new entities can be easily incorporated into the language by virtue of the modular structure that the O-O approach supports. The syntax for such a declaration is as follows:

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

The declaration block is bound by the key words *Devices* and *End*. The *<device_variable>* is a user-defined alphanumeric string with possible underscores separating the individual characters in the strings. The *<device_type>* is one of the pre-defined key words which corressponds to a actual physical device. The *<port_variable>* is a label associated with a physical port that is assumed to have been declared previously in the port declaration section. The *<bit_number>* field is a numeric value between 0 and 7 which corressponds to a physical bit on the communication port. It is this bit that controls the manufacturing device, partly or wholly. In cases where this bit is associated with input, its state represents the state of the device associated with it. The example below illustrates the syntax of this declaration.

```
Devices
        PalletLiftUp    pulse                   PortC 4
        Conveyor        Coil                    PortC 5
        Robot           Programmable LPT1
End
```

The table below lists the device types and their associated functions.

Device Type	Valid Functions
Coil	On, Off
Sensor	Waiton, Waitoff
Pulse	Strobe
Programmable	Send, Do
Wait	milliseconds

Table 3.1.2.1: Device functions

It is worthwhile to note that the programmable device type represents the class of manufacturing devices that can be programmed by a set of primitive instructions that are unique to the particular device class. It is in fact this type which hides this detail away from the user.

3.1.3.Cell Declarations

This subset of declarations associates each manufacturing cell with a computer which controls the functioning of that particular cell. This is done to facilitate inter-cell communication through the underlying computer network. The syntax is shown below.

```
<cell_variable>          <computer_id>
```

Here *< cell_variable >* associates a manufacturing cell with a character string. The terminal *< computer_id >* is a network address that uniquely identifies a computer on the network. Addresses can be either Internet addresses or names that uniquely map to an Internet address. Two examples below explain the syntax. These declarations are contained within the *Cells...End* block. The experimental CIM configuration that we have developed as a part of this project has one cell at present. However, additional cells can be easily added to this.

```
Cells
        Cell1          Cell1_Comp
        StorageCell    134.53.32.240
End
```

3.1.4.Procedure statements

A CPL file comprises the following units:

1. Procedures: Contains the instruction sequence to control and operate cell.
2. Program: Collection of procedures.

The syntax for procedure statements is shown below:

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

< device_variable > is an identifier previously declared in the device declaration section. The < device_function > is a key word defined in the language. The function(s) associated with each device are listed in table 3.1.2.1. All statements are within a *Procedure...End* block. The following is an example of the syntax for procedure statements.

```
Procedure
        Conveyor.On;
        Robot.Send (''NT");
        Lathe.Do(MachinePart);
        Delay.500
End;
```

Here again we notice the implementation-independent syntax (and semantics) of the instructions. The final "container" for all the language entities described earlier is the *program* construct. This is essentially a collection of statements that invoke procedures which are assumed to have been declared before. This approach enforces a modular structure to the programs, thus enhancing readability and promoting ease of development and maintenance. As before, all program statements are enclosed by *Program...End* block. The syntax of a program statement is as follows:

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

One important observation about this syntax is the presence of an optional clause that, in effect, permits repetition. This can be explicitly specified by number of iterations or a condition so that repetition occurs as long as the boolean value of the expression remains unchanged. This condition is a special one in that it refers to a signal from a cell controlling computer. The idea behind this can be explained by a simple example. Consider a series of cells along a manufacturing line for cars. Let us assume that two adjacent cells machine the engine block. Further, let us assume that the first cell performs a milling operation on the engine block and the subsequent one does a finishing operation on the blocks. Now, we would like the second cell to perform its operations as long as there are parts arriving from its successor cell. Another way of stating this is to say that this cell repeats its functions as long as the previous cell is up and running. It is interesting to note that this approach blends well with the declarative style of program-

ming in which we specify what we want as opposed to how to do it. In other words, this is is yet another instance of the abstraction that we have been constantly emphasizing [11], [12]. At this point this feature is a part of the language definition, but the present version of its implementation does not support it. A final example illustrates this syntax of the program statement.

```
Program
        MachinePart.EngineBlockCell; *This is a comment!
        StoreParts.StorageCell (ManufacturingCell.SignalOn);
        StoreParts.RetrieveParts(50);
End
```

The above example illustrates the way in which comments are inserted. In CPL, every statement except block marker sends with a semicolon. An example of a CPL program is included in the appendix B. A copy of the *user manual* can be found in appendix C.

3.2.Construction of the Interpreter for Executing CPL

The development of tools to translate and execute CPL instructions was done in two phases. In the first phase, all the CPL code is compiled to produce an intermediate representation, which we called *I-code representation*. This representation resembles any standard assembler syntax. The reader is urged to refer to appendix D for a sample of this representation. In the second phase, the I-code file is interpreted and commands are generated for the hardware interface which controlled the manufacturing cell. The reason for this two-tiered approach can now be seen in the light of CNC approach which was discussed in the introduction. It is the interpreter which provides the control of the cell (and hence the peripherals that make up the cell), to the computer controlling it. Thus have we moved the control from the device controllers up into the software layers. This enhances the flexible nature of the manufacturing system in that it lets us control the functioning during the actual execution of the instruction. The development of the tools for the first phase of the translation, i.e., the compiler and the cross-reference listing generator was taken up by another graduate student [13]. The figure below summarizes these two stages of language

translation process.

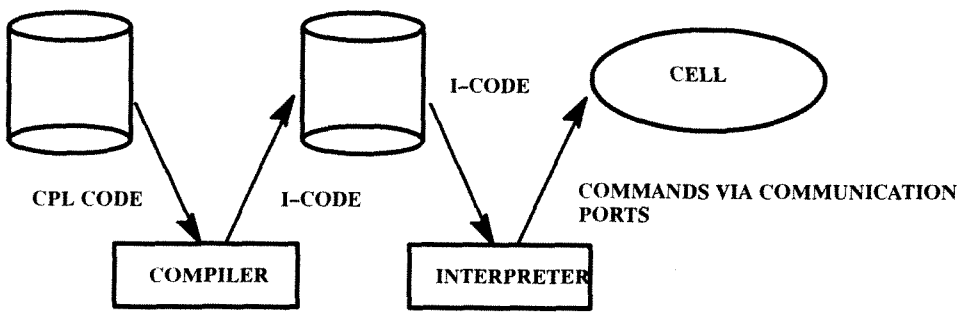


Table 3.2.1: Two stage CPL language translation process

3.2.1.Functional Description of the CPL Interpreter

The interpreter serves the following services:

1. It enforces control of the manufacturing cell in the controlling computer by executing one instruction at a time.

2. It optionally allows user interaction and thus passes the control to the user level. This is done by providing execution under *step (-s)* mode. This mode outputs the CPL command and waits for the user prompt before executing the corresponding set of low-level instructions. In other words, the interpreter "steps" through instructions one at a time in an asynchronous fashion. This option also allows provides runtime debugging facilities.

3. An option to *trace (-t)* is also provided to enable the user to associate each CPL instructions with the actual physical action that the instruction represents. This is provided to make the environment more user-friendly.

With these specification of functional requirements, the next step was to design the structure of the interpreter. It was decided to separate the interpreter from the compiler component from a pure software engineering perspective. This would result in decoupling and thus code inter-dependencies would

be totally eliminated. It was again decided that the O–O approach would be suitable for reasons already discussed in the previous sections of the paper. The design aspects are explained in the next section.

3.2.2.Design and Implementation of the Interpreter

The first step in the design process was to identify the tasks that the interpreter had to carry out. It should be recalled that the interpreter had to execute intermediate code, called *p–code*, that is output by the compiler which takes *CPL* source as input. With this in mind, five action primitives were recognized which are listed below:

(*SetBit*) Set or reset a bit on the I/O port.

(*QueryWait*) Wait for the system to enter a particular state.

(*Wait*) Wait unconditionally for a specified amount of time.

(*SendString*) Send a text string over an interface.

(*Strobe*) Send a strobed signal via the I/O port.

In addition, actions pertaining to setting up of the ports and debugging features were recognized. The actions listed above will be explained in detail now.

Every manufacturing device is associated with one or more bits on the I/O port, the number of bits being directly related to the number of abstract devices that a physical manufacturing device handles. For example, a CNC lathe would be considered equivalent to three logical devices, the first one to switch it on or off, the second one to toggle it into a mode to accept instructions via the serial interface, and the third one to control the opening and closing of the chuck (This is not entirely true since the chuck can be considered as an independent device and thus be decoupled from the lathe, which is yet another manifestation of the o–o paradigm that we have employed!).

Often, it is necessary to wait for the system to enter a particular state before the next action can be taken. Consider another situation which employs the second action primitive to achieve this goal. Before the chuck can release the part that has just been machined by the lathe, it is very important that the robot grasps the part first, otherwise we will have the robot in a fairly embarrassing situation of not knowing how to get hold of the part! Therefore, the face saving measure here would be to wait until the robot has grasped the part with its gripper before the chuck can be asked to release the part. So by employing the *QueryWait* action primitive, it is possible to make the chuck wait till the robot grasps the part following

which it can safely release the part. This is illustrated by a generalized state diagram shown below.

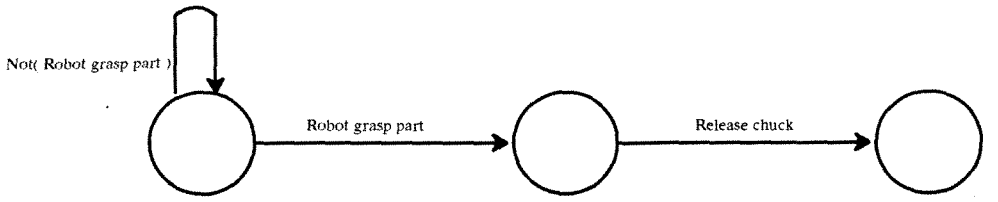


Figure 3.2.2.1: State transition diagram for QueryWait action primitive

The third action primitive, *Wait*, is similar to the previous one but does not require the constraint of having to wait for the system to get into a particular state. This unconditional wait primitive allows the user to decide on the wait time between actions. Typically, this is useful when sending a sequence of port initializations with a fixed time interval between individual initialization. This is often necessary when a user wants to be sure that the system has reached a given state before taking the intended action.

The *SendString* primitive primarily addresses the issue of sending text strings to programmable devices such as robot and lathe over an appropriate interface. These strings are commands that the devices understand and execute. In fact, one of the aims of *CPL* was to hide this relatively primitive set of commands from the end-user as far as possible. This primitive achieves this as follows: Files containing these commands are generated once for a particular setup of the cell and stored away in a database residing on the network server. (Details about the network are explained in section 4). By using the *SendString* primitive, the user can request the downloading of these programs to specific programmable devices.

The last primitive action, *Strobe*, is identical to *SetBit* except that this action sends a step input (a high followed by a low on the electrical interface) to the I/O port. This action is to accommodate certain manufacturing devices that require a strobed input for activation or deactivation.

With this identification of tasks that the interpreter, the next step was to identify the data elements that would be necessary. In formal o-o terms, this meant the identification of the *nouns* as objects, the *verbs* being the verbs that have just been discussed. The inheritance structure is illustrated in appendix E. The code implementing the interpreter is included in appendix F.

4.Network based Tools for Management and Control of Manufacturing Cells

4.1.A Brief Introduction to Manufacturing Networks

The role of a computer based network is very critical to the functioning of a CIM setup. It is the network which provides means of communication between various entities, as well as integrate the design, manufacturing and management components successfully. So a reliable and a timely network is very essential in the CIM world. A hierarchical model for a network can be thought of comprising the following levels:

1. Factory level network.
2. Shop level network.
3. Cell level network.
4. Machine level network.
5. Sensor level network.

It is interesting to note the close correspondence of this hierarchy with the structured layers that *CPL* provides. The performance variables for each of these layers can vary depending on the requirements of the network at each level. The same is true abut the topology of the network at each of these levels. The table below summarizes this information.

Network level	Topology	Medium access control	Performance variables	Overall functionality
Factory	Partial interconnections	Point to point	Throughput	High/Medium
Shop	Bus, Ring, Loop, Star	Token passing, Ethernet	Throughput	High
Cell	Bus, Ring, Loop, Star	Token passing, Ethernet	Response time, Throughput	High
Machine	Bus	Token passing	Response time	Medium
Sensor	Bus	Polling, Token passing	Response time	Low

Table 4.1: Hierarchical network structure and performance variables for Manufacturing

As can be seen, the performance is measured by response time as we move down the hierarchy. This is consistent with the fact that the lower levels of the network hierarchy are working at the machine level, and hence the real-time performance of such networks would be of critical importance.

4.2.A Structured Approach for Designing Manufacturing Networks

There are a number steps involved in the design of a manufacturing network. The following are significant:

1. Selection of network architecture.
2. Design of a specific network topology.
3. Choosing specific implementations.
4. Development of a set of application requirements.
5. Translation of application requirements into a set of network requirements.
6. Design of application software.
7. Evaluation of the overall design.

The selection of an architecture depends largely on the functionality. Stringent real time requirements would imply a fast and a reliable network which is characteristic of lower two layers of the hierarchy listed before. As can be seen, Ethernet is not preferred in the lower levels, since its performance seems to degrade with increasing traffic [14]. Token bus architecture is preferred in these layers.

Topology is largely governed by the possibility of further expansion of the existing network. Whenever possibilities of expansion exist, a star topology is preferred since it is very easy to add additional nodes on such a setup. Referring to table 4.1, we see that star configuration is a topological option in the top three layers whereas it is absent in the lower ones. This is due to the fact that at the machine and the sensor levels, there is very little possibility of additions being made to the network.

Since there are many protocol and network implementations, network application designers must choose specific implementations so that they match the selected architecture, provide adequate performance, and easily interface with other devices used in the application. Factors affecting implementation include operating systems, programming languages, etc.

Definition of application requirements is a statement of what the manufacturing systems needs to deliver. This is done to extract the implications of such requirements on the performance of the network component of the manufacturing system.

Once the application requirements are clearly defined, the next stage would be to make some decisions about the network requirements. The important factors that come into play here are response time,

information throughput, medium type for all networks and subnetworks, tolerance of failures, software interface, etc.

Once the network requirements of the network are defined, the next step would be to design application software to run on such a network. These applications typically perform end user functions. A typical example in this category would be a set of software programs that remotely monitor the status of a cell. Another example could be software that manages a centralized database on the network.

Evaluation of the network is necessary to verify and validate the design. Typically, this can be performed using analytical models, simulation or measured data.

4.3.A Prototype Network for CIM Lab at Miami University

The network that was designed and installed at Miami university spanned the factory, and shop levels of the network hierarchy. As a result, a combination of bus and star topology was chosen. Since the traffic on the network was expected be low, an Ethernet was chosen. This also rendered the network compatible with the existing networks in the building as well as the networking software that is currently being used. The configuration of the network is shown in appendix G. A star topology using twisted pair cables was chosen at the cell shop level to accommodate for additional cells in the future and also allow the relocation of cells. A bus topology was employed to connect the computers that were identified to be used in the design and the analysis components of the CIM model.

4.4.Network Applications

The network applications that were developed were to add remote monitoring capabilities to the network so that an user could monitor the status of a cell from any computer that was on the network. The Remote Procedural Call toolkit was utilized for this purpose which provided some ready to use program shells that could be customized for specific applications.

The client server approach was taken to develop this application. It was decided that the client would run on the computer that also controlled the cell, along with the interpreter which drove the cell. This posed a problem since the operating system that runs on these computers does not have the capability to support multitasking. This shortcoming was overcome by using *Terminate and Stay Resident (TSR)* technique. This approach essentially hooks an application program to one of the system interrupts (like keyboard or timer interrupt) and activates the application whenever an interrupt is generated. It is up to the

application to take some action or pass the control over to the default interrupt handler. Thus, using this approach, the client program that monitors the status of the cell runs in the background and the interpreter runs in the foreground. The client monitors the status by observing any changes on the I/O port to which all the manufacturing peripherals are hooked. On the server side, a program that polls the network for updated information about the cell, is kept running. If any updated information is received, the display screen is updated to reflect these changes.

One interesting real time constraint emerged during the development of this application. The client that monitors the cell can be set to poll the system at fixed time intervals. If these time intervals are spaced too far apart, then it is possible to lose transitions that occur in the system between the two check points. For example, it is possible that the photo sensor detects a pallet, sets a bit on the I/O port, then resets it after the pallet passes, before the monitoring program checks the status again. This loss of information can result in inaccurate status information. On the other hand, polling the I/O port too often would degrade the performance of the interpreter and could the affect the performance of the cell. Thus, a trade off had to be made by trial and error to decide on an acceptable time interval between successive pollings. The code that implements the monitoring system is included in the appendix.

5.Summary and Future Directions

This project attempted to analyze the various software and computer requirements for a computer integrated manufacturing environment. The development of *CPL* addressed the language issues and the present version of it has been successfully tested for correctness. Due to time and resource constraints, some issues could not be considered in this project. Firstly, a truly multitasking operating system with some real time capabilities is needed for a CIM environment. Presently, the operating system, DOS, supports single tasking only with practically no real time capabilities and this is a serious bottleneck in the system. *CPL* needs to be enhanced to support shared memory capability, thus allowing multiple processes to access a common memory area (which is necessary when two distinct processes coordinating with each other). Shared memory places additional burden on the application languages and the operating system, since now one has to make sure that memory updates by one process are not overwritten by the other processes. This needs memory lock mechanism like semaphores. As the number of cells on the shop floor increase, it is necessary to build in concurrency into the software system to allow for the network hierarchy

to be extended to the factory layer. This would enable a central computer to coordinate various cells operating in parallel. Another feature that is needed is priority scheduling of processes, and this needs to be passed on to the application layers since the user has control over assigning priorities to different processes.

Attempts have been made to preserve the declarative style of *CPL* as far as possible. At this point, the language does not support constructs like *if–then–else*, *repeat–until*, *while–do*, etc. Although these constructs make a language more powerful, it should be noted that a certain amount of procedural flavor is introduced into the language. On further consideration, it seems that this is necessary to deal with real time constraints of the system. Another possibility can be recursion to provide declarative form of looping, but this generally degrades the performance at runtime. So it can be concluded that these constructs have to be incorporated at the cost of losing some of the declarative flavor of the language.

Another important issue that needs to be considered is the extension of the language to offer network support. This is necessary if the language needs to support multiple cell management in a network transparent way. This would make it unnecessary for the user to know if two processes are resident on the same machine or on different machines. This would offer a client–server model which can be conveniently used to implement such applications as remote monitoring applications. Also, this would make it possible to establish a homogeneous language interface across the design and manufacturing components of the CIM model since these components generally reside on different machines on the network.

The CIM model that has been developed does not yet incorporate an integrated user interface, which is necessary to provide a common software platform that spans the various stages of an integrated environment. This would provide a window–based, menu–driven interface to the user that could hide all the network details from the user and provide an easy to use interface.

In conclusion, *CPL* provides a platform to develop simple programs by the students intending use it. Peripherals can be added to and deleted from the cell very easily by incorporating appropriate device declarations. The interpreter is dependent on the hardware to a certain extent, but this unavoidable. Any new action primitives can be very easily added to it since its object–oriented nature makes it highly modular. The remote monitoring facility is very simple at this stage, but the basic framework has been established to facilitate further enhancements to it.

References

1. Yoran Koren, *Computer Control of Manufacturing Systems*, McGraw Hill, 1983.
2. Hassan Gomma, *A Software Design Method for Real Time Systems*, Comm. of the ACM, September 1984.
3. Hassan Gomma, *Software Development of Real Time Systems*, Comm. of the ACM, July 1986.
4. Brian M. Barry, *Smalltalk as a Development Environment for Integrated Manufacturing Systems*, Proceedings of the International Conference on Object-Oriented Manufacturing Systems, May 1982.
5. Ian Sommerville, *Software Engineering*, Addison-Wesley, 1992.
6. Nof S. Y., *Is all Manufacturing Object-Oriented?*, Proceedings of the International Conference on Object-Oriented Manufacturing Systems, May 1982
7. Fauvel, et. al., *Object-Oriented Design for Manufacturing*, Proceedings of the International Conference on Object-Oriented Manufacturing Systems, May 1982
8. Prabhakar S., et. al., *An Evaluation of Object-Oriented Design Methodologies for Control and Manufacturing Environments*, Proceedings of the International Conference on Object-Oriented Manufacturing Systems, May 1982.
9. Ken Jenne, Pat Pascal, *A Software Architecture for Building Industrial Automation Systems*, Proceedings of the International Conference on Object-Oriented Manufacturing Systems, May 1982.
10. John C. Kelly, *A Comparison of Four Design Methods for Real Time Design*, Proceedings of the Ninth International Conference on Software Engineering, Monterey, CA, March 1987.
11. Des Watson, *High-Level Languages and their Compilers*, Addison-Wesley, 1989.
12. Ravi Sethi, *Programming Languages- Concepts and Constructs*, Addison-Wesley, 1989.
13. Mala's Report ??
14. Juan R. Pimentel, *Communication Networks for Manufacturing*, Prentice-Hall, 1990.
15. Sun Microsystems, *PC-NFS Programmers Toolkit*.

Appendix A
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 are as shown below.

CPL Production Rules

1. < cpl_program > → PROGRAM < prog_name > < declarations >
2. < declarations > → [< port_declarations >] < device_declarations > < procedure_declarations >
3. < port_declarations > → PORTS < port_stmntList > END
4. < device_declarations > → DEVICES < device_stmntList > END
5. < procedure_declarations > → PROCEDURE < procedure_stmntList > END
6. < port_stmntList > → < port_stmnt > [< port_stmntList >]
7. < port_stmntList > → < port_stmnt >

8. < port_stmnt > → < port_name > < port_address > < direction >;
9. < port_name > → < identifier >
10. < port_address > → < integer >
11. < device_stmntList > → < device_stmnt > [device_stmntList >]
12. < device_stmntList > → < device_stmnt >
13. < device_stmnt > → < device_name > < device_type > < device_data >;
14. < device_name > → < identifier >
15. < device_type > → PROGRAMMABLE | < nonpgble_type >
16. < device_data > → { (< port_name > < valid_bit >) , < predefined_port > }
17. < procedure_stmntList > → < procedure_stmnt > [< procedure_stmntList >]
18. < procedure_stmntList > → < procedure_stmnt >
19. < procedure_stmnt > → < sequence > | < repetition > | < alternation >
20. < sequence > → < device_name > [. < dev_func >] ;
21. < *repetition* > → DO < iterations > < procedure_stmt_list > END DO
22. < dev_func > → < function >
23. < function > → < pulse_func > | < coil_func > | < sensor_func > | < programmable_func > | < wait_time >
24. < pulse_func > → . Strobe
25. < coil_func > → . { On , OFF }
26. < sensor_func > → . { WaitOn , WaitOff }
27. < programmable_func > → . Send < open_parenthesis > { (< double_quote > < string > < double_quote >) , < identifier > } < close_parenthesis >
28. < wait_time > → . < integer >
29. < direction > → [INPUT] |OUTPUT
30. < nonpgble_type > → COIL|SENSOR|PULSE|WAIT
31.. < predefined_port > → LPT1:|COM2:
32. < valid_bit > → 0|1|2|3|4|5|6|7
33. < iterations > → < integer >
34. < identifier > → < string >
35. < string > → < character > [< character >]
36. < character > → < alphabet > | < special_character > | < digit >
37. < alphabet > → A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
38. < special_character > → < parenthesis > | < braces > | < flower_bracket > | < math_operator > |'| ~ |!|@|#|$|%|^|&|*|(|)|_|:|"|,|.|?|/|\|';
39. < double_quote > → "
40. < parenthesis > → < open_parenthesis > | < close_parenthesis >
41. < braces > → < open_brace > | < close_brace >
42. < flower_bracket > → < open_flower_bracket > | < close_flower_bracket >
43. < open_parenthesis > → (
44. < close_parenthesis > →)
45. < open_brace > → [
46. < close_brace > → [
47. < open_flower_bracket > → {
48. < close_flower_bracket > → }
49. < math_operators > → +|-|*|/
50. < integer > → < digit > { < digit > }
51. < digit > → 0|1|2|3|4|5|6|7|8|9

Appendix B
CPL Source Program

```
Ports           /* Port declarations
        PortC   64259 Output;
        PortA 64256 Input;
End

Devices         /*  Device declarations
        PalletLiftUp    Pulse   PortC  4;
        Conveyor                Coil            PortC  5;
        PhotoCell               Sensor          PortA 7;
        PalletArrived   Sensor  PortA 6;
        ChuckOpen       Pulse   PortC 1;
        Robot           Programmable LPT1;
        LatheStart              Pulse           PortC 2;
        LatheStop               Sensor          PortA 4;
        PalletLifted            Sensor          PortA 5;
        PalletStops             Coil            PortC 0;
        ChuckClose              Pulse           PortC 3;
        PalletLiftDownPulse     PortC 6;
        LatheRunning Sensor     PortA 2;
        LatheHandShk            Sensor          PortA 3;
        Delay                   Wait;
End

Procedure               /* Device operations
        Robot.Send("NT");
        PalletStops.On;
        Conveyor.On;
        PhotoCell.WaitOn;
        PalletStops.Off;
        PalletArrived.WaitOn;
        Delay.1000;
        PalletLiftUp.Strobe;
        Pallet.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;
```

Appendix C
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. Data Structures 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 structures are composite data types and a declaration of a variable to be of a type also assigns values to the attributes of the type. So, a dynamic change in the value of a variable 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 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 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.

Appendix D
I-Code Representation

Note: The italicized text represents the CPL statement whose I-code representation appears below.

11 ComPort COM1: 300 7 2 0
9 64 2 4 0 0
11 PortA 64256 Input
10 64256 input
11 PortB 64257 Output
10 64257 output
11 PortC 64258 Output
10 64258 output
11 LatheHandshake.Strobe
7 64257 0 1
11 LatheG66inp.Strobe
7 64257 1 1
11 Lathe.Do(loadlathe)
8 COM1 %

8 COM1 N' G' X ' Z ' F' H

8 COM1 00M03

8 COM1 01 00 00 - 7100

8 COM1 02 01 - 100 00 80

8 COM1 03 01 - 50 50 25

8 COM1 04 01 00 500 25

8 COM1 05 01 50 50 25

8 COM1 06 00 100 00

8 COM1 07 00 00 6500

8 COM1 08M05

8 COM1 09M00

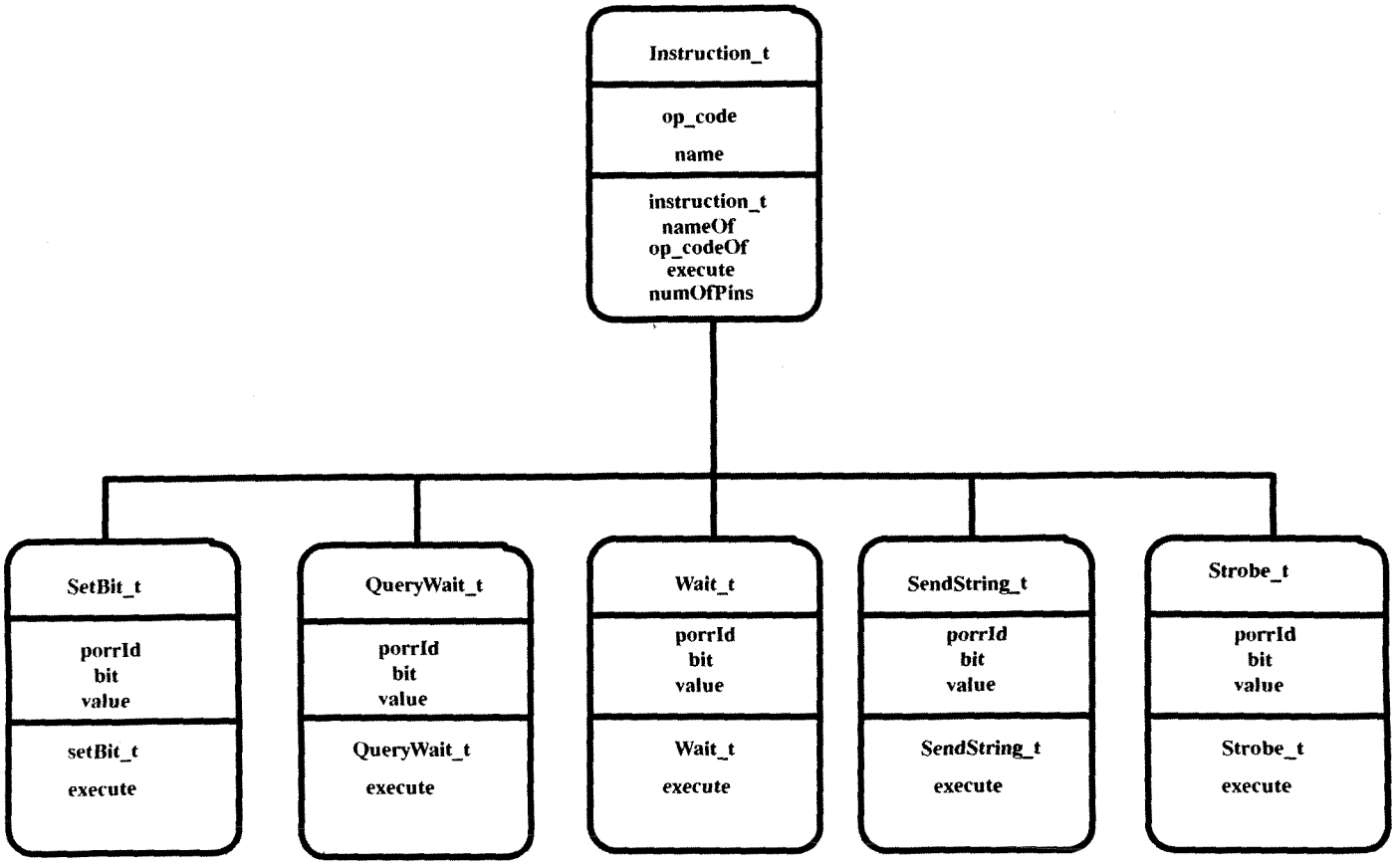
8 COM1 10M30

8 COM1 "
11 Robot.Send("NT")
5 LPT1 NT
11 PalletStops.On
1 64258 0 1

```
11          Conveyor.On
1 64258 5 1
11          PhotoCell.WaitOn
3 64256 7 1
11          PalletStops.Off
2 64258 0 0
11          PalletArrived.WaitOn
3 64256 6 1
11          Delay.1000
6 1000
11          PalletLiftUp.Strobe
7 64258 4 1
11          PalletLifted.WaitOn
3 64256 5 1
11          Conveyor.Off
2 64258 5 0
11          ChuckOpen.Strobe
7 64258 1 1
11          Robot.Do(loadpart)
8 LPT1 MI -2400,-1600,800,1570,1390,0
8 LPT1 MI 0,-240,-540,225,-225,0
8 LPT1 GC
8 LPT1 MI 0,1020,-240,-55,55,0
8 LPT1 MI -7200,200,1000,0,0,0
8 LPT1 MI 250,-1800,1700,0,0,0
8 LPT1 MI 20,-530,175,0,0,0
8 LPT1 MI -130,0,0,0,0,0
11          Delay.1000
6 1000
11          ChuckClose.Strobe
7 64258 3 1
11          Delay.2000
6 2000
11          Robot.Do(moveaway)
8 LPT1 GO
8 LPT1 MI 130,0,0,0,0,0
8 LPT1 MI -20,530,-175,0,0,0
8 LPT1 MI 0,2360,-2660,-1680,-1160,0
11          Delay.2000
6 2000
11          LatheStart.Strobe
7 64258 2 1
11          LatheStop.WaitOff
4 64256 4 0
11          Robot.Do(MoveBack)
```

```
8 LPT1 MI -250,-560,960,1680,1160,0
8 LPT1 MI 250,-1800,1700,0,0,0
8 LPT1 MI 20,-530,175,0,0,0
8 LPT1 MI -130,0,0,0,0,0
8 LPT1 GC
11          Delay.2000
6 2000
11          ChuckOpen.Strobe
7 64258 1 1
11          Delay.2000
6 2000
11          Robot.Do(GetPart)
8 LPT1 MI 130,0,0,0,0,0
8 LPT1 MI -20,530,-175,0,0,0
8 LPT1 MI -250,1800,-1700,0,0,0
8 LPT1 MI 7200,0,0,0,0,0
8 LPT1 MI 0,-1180,-760,55,-55,0
8 LPT1 GO
8 LPT1 MI 2300,1700,-160,-1695,-1265,0
8 LPT1 NT
11          PalletStops.On
1 64258 0 1
11          PalletLiftDown.Strobe
7 64258 6 1
11          Conveyor.On
1 64258 5 1
11          Delay.500
6 500
11          Conveyor.Off
2 64258 5 0
11          PalletStops.Off
2 64258 0 0
11          LatheStart.Strobe
7 64258 2 1
11          LatheHandshake.Strobe
7 64257 0 1
```

Appendix E
Class Hierarchy for the Interpreter



Appendix F
Code for CPL Interpreter

```
//-----[ instruct.h ]-----------------------------------------

// Instruction class hierarchy

#ifndef INSTRUCT_H
#define INSTRUCT_H

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

#define SIZE            80
#define TRUE    1
#define FALSE   0

class instruction_t {
   int    op_code;
   char   name[21];
public:
   instruction_t(int op=0, char *n="Noop") { op_code = op; strcpy(name,n); }
   int isA(char *);
   char *nameOf(void);
   int  op_codeOf(void);
   virtual int execute(void) = 0;
   virtual int numOfPIns(void){return(1);};
};

class cplSource : public instruction_t {
private:
            int pCodeInst;
            char *cplCode;
public:
            cplSource(int pCode, char *cplBuffer);
            int numOfPIns(void);
            int execute(void);
};

class setBit_t : public instruction_t {
   unsigned int portid;
   int bit;
   int value;
public:
   setBit_t(FILE *fp);
   int execute(void);
};
```

```
class queryWait_t : public instruction_t {
   unsigned int portid;
   int bit;
   int value;
public:
   queryWait_t(FILE *fp);
   int execute(void);
};

class wait_t : public instruction_t {
   int millisec;
public:
   wait_t(FILE *fp);
   int execute(void);
};

class sendString_t : public instruction_t {
               char port[16];
               char text[80];
   static int  flag;
public:
   sendString_t(FILE *fp);
   int execute(void);
};

class strobe_t : public instruction_t {
   unsigned int portid;
   int bit;
   int value;
public:
   strobe_t(FILE *fp);
   int execute(void);
};

class commSetup: public instruction_t {
            unsigned char          baudRate;
            unsigned char   dataBits;
            unsigned char          stopBits;
            unsigned char          parity;
                        int    commPort;
public:
               commSetup(FILE *fp);
            int execute(void);
};

class dabSetup: public instruction_t {
            unsigned int  portid;
```

```
                    char mode[10];
            static  int  portInitialized;
public:
               dabSetup(FILE *fp);
            int execute(void);
};


#endif
```

```
//---[ list.h ]-----------------------------------------------------

#ifndef LIST_H
#define LIST_H

#include "instruct.h"

//
// class definition for a list "selectable" objects.
//
// this is a generic list class -- change the type names and reuse.
//

//
// Change this typedef to make the list operate on other types
//

typedef instruction_t * objptr_t;

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

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
```

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

#ifndef FUNC_H
#define FUNC_H

/* 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 CCOMSETUP       9
#define CDABSETUP 10

#endif
```

```
//---[ instruct.cpp ]-----------------------------------------
#include <stdio.h>
#include <dos.h>
#include <bios.h>
#include <string.h>
#include <conio.h>
#include "opcodes.h"
#include "instruct.h"
#include "list.h"

extern int trace, singleStep;

// Methods for instruction_t

int instruction_t::isA(char *n)
{
   if (strcmp("instruction_t",n) == 0) return 1;
   else return 0;
}

char * instruction_t::nameOf(void)
{
   return(name);
}

int instruction_t::op_codeOf(void)
{
        return(op_code);
}

//Methods for cplSource

cplSource::cplSource(int pCode, char *cplBuffer) : instruction_t (CPL_COMMAND, "cplSource")
{
        pCodeInst  = pCode;
        cplCode = new char[strlen(cplBuffer)+1];
        strncpy(cplCode, cplBuffer, strlen(cplBuffer)+1);
}

int cplSource::numOfPIns(void)
{
        return(pCodeInst);
}

int cplSource::execute(void)
{
        if (trace)
```

```
        {
                clrscr();
                printf("%s", cplCode);
                delay(2000);
        }
        if (singleStep)
        {
        clrscr();
                printf("%s", cplCode);
        printf("\nHit return to continue");
                getch();
                clrscr();
                printf("\nWait ...");
        }
        return(1);
}

// Methods for setBit_t

setBit_t::setBit_t(FILE *fp) : instruction_t (SET_BIT_ON, "setBit_t")
{
   fscanf(fp,"%u %d %d",&portid,&bit,&value);
}

int setBit_t::execute(void)
{
   unsigned char status = inportb(portid);
   unsigned char mask = 1 << bit;

   if (value)
        outportb(portid, status|mask);
   else
        outportb(portid,status&~mask);

#ifdef VERBOSE
   printf("Executed setBit, port = %u, bit = %d, value = %d\n",
                portid,bit,value);
#endif
   return 1;
}

// Methods for wait_t

wait_t::wait_t(FILE *fp) : instruction_t (WAIT, "wait_t")
{
   fscanf(fp, "%d", &millisec);
}
```

```
wait_t::execute(void)
{
   delay(millisec);
   return 1;
}

// Methods for queryWait_t

queryWait_t::queryWait_t(FILE *fp)
{
   fscanf(fp, "%u  %d %d", &portid,&bit,&value);

}

queryWait_t::execute(void)
{
   unsigned char result = inportb(portid);
   unsigned char mask = 1 < < bit;

   if( value = = 1)
            while(!(mask & result)) result = inportb(portid);
   else
            while((mask & result)) result = inportb(portid);

   #ifdef VERBOSE
     printf("queryWait value = %d, bit = %d, port = %d\n",
                        value,bit,result);
   #endif
   return 1;
}

// Methods for sendString_t

sendString_t::sendString_t(FILE *fp)
{

   fscanf(fp, "%s", port);
   fgets(text, sizeof(text), fp);
}

sendString_t::execute(void)
{
   char *ip = text;
   int len;

            if (strcmp(port, "LPT1") = = 0)
      {
                        len = strlen(text);
                        for(ip=text; *ip = = ' ' && *ip != '\0'; ip+ + );
```

```
                text[len-1] = '\r';
                text[len] = '\n';
                text[len+1] = '\0';
                fprintf(stdprn,"%s",ip);
        }
        if (strcmp(port, "COM1") == 0)
        {
                        // set RTS bit. CTS is automatically set on
                        // the null modem.
                        //outportb(0x3fc, 1);
                        len = strlen(text);
                        text[len-1] = '\r';
                        text[len] = '\n';
                        text[len+1] = '\0';

                        // now down load the string contained in text
                        for(int i=0; i<strlen(text); i++)
                        {
                                        while (text[i] == 0x40)
                                                i++;
                                        int mask = 1<<5;
                                        int status = inportb(0x3fd);

                                        // Is the transmitter holding
                                register empty??
                                        while ( !(status & mask))
                                                status = inportb(0x3fd);
                                        //outportb(0x3fe, 0);
                                        outportb(0x3f8, text[i]);

                        }
    }
    #ifdef VERBOSE
                printf("sendString execute, sent %s\n",text);
    #endif
    return 1;
}
// Methods for strobe_t

strobe_t::strobe_t(FILE *fp) : instruction_t (STROBE, "strobe_t")
{
   fscanf(fp,"%u %d %d",&portid,&bit,&value);
}

int strobe_t::execute(void)
{
```

```
    unsigned char status = inportb(portid);
    unsigned char mask = 1 < < bit;

    if (value)
    {
            outportb(portid, status|mask);
            delay(500);
            outportb(portid, status& ~ mask);
            delay(500);
    }
    else
    {
            outportb(portid, status& ~ mask);
            delay(500);
            outportb(portid, status|mask);
            delay(500);
    }
#ifdef VERBOSE
    printf("Executed strobe, port = %u, bit = %d, value = %d\n",
                        portid,bit,value);
#endif
    return 1;
}

// Methods for commSetup

commSetup::commSetup(FILE *fp) : instruction_t (COMM_SETUP, "commSetup")
{
    fscanf(fp,"%d %d %d %d %d",&baudRate,&dataBits,&stopBits, &parity, &commPort);

}

int commSetup::execute(void)
{
   char settings = baudRate|dataBits|stopBits|parity;

   bioscom(0, settings, commPort);

#ifdef VERBOSE
    printf("Executed commSetup, baudRate = %d, dataBits = %d, stopBits = %d, parity = %d,
commPort = %d\n",
                        baudRate,dataBits,stopBits, parity, commPort);
#endif
    return 1;
}

// Methods for dabSetup
```

```
dabSetup::dabSetup(FILE *fp) : instruction_t (DAB_SETUP, "dabSetup")
{
   fscanf(fp,"%u %s",&portid,mode);

}

int dabSetup::execute(void)
{

  /*************************************************************
   * IMPORTANT! The following piece of code sets up the      *
   * Intel 8255A I/O chip with ports B, C for output         *
   * and port A for input. Any changes or replacements done *
   * in the future should corresspondingly be made here.    *
   * Please consult the hardware manual if in doubt.         *
   *************************************************************/

  outportb(portid, 0);
  if(!portInitialized)
  {
          outportb(64259,144);
          portInitialized = TRUE;
  }

#ifdef VERBOSE
   printf("Executed dabSetup, portid = %u, mode = %s\n", portid, mode);
#endif
   return 1;
}
```

```
// ---[ interprt.cpp ]-----------------------------------

#include <stdio.h>
#include <dos.h>
#include <bios.h>
#include <string.h>
#include <conio.h>
#include "opcodes.h"
#include "instruct.h"
#include "list.h"

char cplBuffer[SIZE];
FILE *InputFile;
char *FileName;
int  trace = FALSE;
int  singleStep = FALSE;

int get_instructions(list_t *ilist, list_t *sourceList)
{
   instruction_t *iptr;
   int opcode, pCode;
   int FirstTime = FALSE;

   if((InputFile = fopen(FileName, "r")) == NULL)
   {
            printf("\nInput file not found");
            return(0);
   }
   while (fscanf(InputFile,"%d",&opcode) == 1)
   {
            switch(opcode)
            {
             case COMM_SETUP:
                ilist->append(new commSetup(InputFile));
                pCode++;
                break;
             case DAB_SETUP:
                ilist->append(new dabSetup(InputFile));
                pCode++;
                break;
             case SET_BIT_ON:
             case SET_BIT_OFF:
                ilist->append(new setBit_t(InputFile));
                pCode++;
                break;
```

```
            case QUERY_WAIT_ON:
            case QUERY_WAIT_OFF:
               ilist->append(new queryWait_t(InputFile));
               pCode++;
               break;
            case SEND_STRING:
            case DO:
               ilist->append(new sendString_t(InputFile));
               pCode++;
               break;
            case WAIT:
               ilist->append(new wait_t(InputFile));
               pCode++;
               break;
            case STROBE:
               ilist->append(new strobe_t(InputFile));
               pCode++;
               break;
            case CPL_COMMAND:
                     if(FirstTime)
                     {
                              fgets(cplBuffer, SIZE, InputFile);
                              pCode = 0;
                FirstTime = FALSE;
                              break;
                     }
                     sourceList->append(new cplSource(pCode, cplBuffer));
                     pCode = 0;
                     fgets(cplBuffer, SIZE, InputFile);
                     break;
            default:
               printf("Undefined opcode %d\n",opcode);
         }
   }
   // append the last piece of source code.
   sourceList->append(new cplSource(pCode, cplBuffer));
   return 1;
}

int execute_instructions(list_t *ilist, list_t *sourceList)
{
   instruction_t *iptr = ilist->first();
   instruction_t *sourcePtr = sourceList->first();
   while (iptr != NULL) {
         if(trace|singleStep)
```

```
                    sourcePtr->execute();
            for(int i=0; i<sourcePtr->numOfPIns(); i++)
            {
                    if(!iptr->execute())
                            return(0);
                    iptr = ilist->next();
            }
            sourcePtr = sourceList->next();
    }
    return 1;
}

main(int argc, char *argv[])
{
    int status;
    list_t *ilist = new list_t;
    list_t *sourceList = new list_t;

    FileName = new char[strlen(argv[1])+3];
    strcpy(FileName, "s:\0");
    switch(argc)
    {
            case 1:
                    printf("Standard usage: interprt [/debug] <file_name>\n");
                    return(0);
            case 2:
                    strncat(FileName, argv[1], strlen(argv[1])+1);
                    break;
            case 3:
                    if ( strcmpi(argv[1], "-t\0") == 0)
                            trace = TRUE;
                    else
            if ( strcmpi(argv[1], "-s\0") == 0)
                                    singleStep = TRUE;
                            else
                            {
                                    printf("\nunknown option");
                                    return(0);
                }

                    strncat(FileName, argv[2], strlen(argv[2])+1);
                    break;
    }
    if (get_instructions(ilist, sourceList))
            status = execute_instructions(ilist, sourceList);
            if (!status)
```

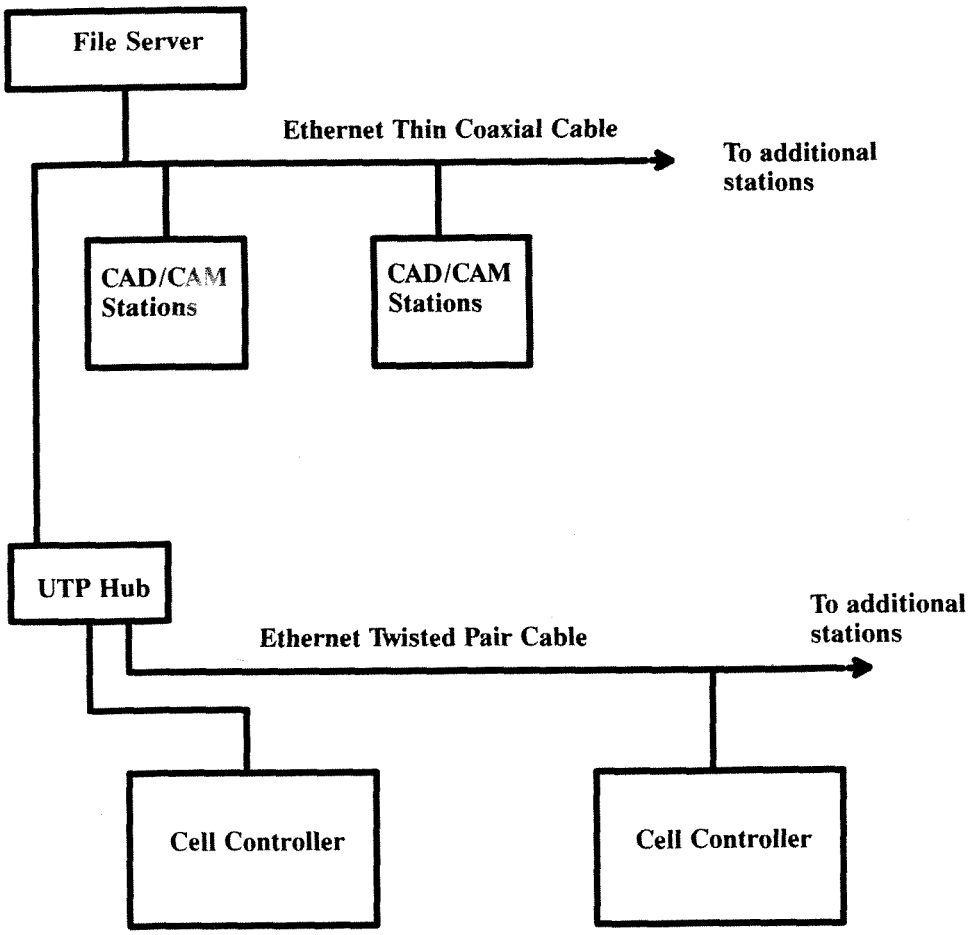
```
        {
                clrscr();
                printf("\naborting at users request...\n");
                return(0);
        }
        return 0;
}
```

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

//
// Member functions for the list class.  This list class has the
// property that elements are always added at the head.  This list
// only contains objects of class "selectable".
//

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

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;
    }

list_t::~list_t(void) {
    entry_t *temp;

    cursor = head.next;

    while (cursor != &tail) {
            temp = cursor;
            cursor = cursor->next;
            delete temp;
            }

    return;
    }

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;
    }
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;
    }
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;
    }
objptr_t list_t::first(void) {

    cursor = head.next;

    return cursor->obj;
    }
objptr_t list_t::next(void) {
```

```
    if (cursor != &tail) cursor = cursor->next;

    return cursor->obj;
    }
objptr_t list_t::last(void) {

    cursor = tail.prev;

    return cursor->obj;
    }
objptr_t list_t::prev(void) {

    if (cursor != &head) cursor = cursor->prev;

    return cursor->obj;
    }
int list_t::length(void) {

    return n_entries;
    }
// end of file
```

Appendix G
Schematic Representation of the Network to Support CIM



```
//-----------------------------------------[ 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**** */

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

/* NAME:         port_t::port_t
   TYPE:                 Class constructor

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

   INPUT PARAMETERS:    token_t*                                          the curr
ent token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                          void

        CLIENT FUNCTIONS:                  translate_t::parsePorts
                                                        table::search

        SERVICING FUNCTIONS: token_t::Type
                                                        token_t::Next
                                                        token_t::Keytype
```

```
                                                                token_t::Identif
ier
                        token_t::Thisline

                                                                error_t::readerr
or
                        error_t::checkerrors

                                                                SYSTEM
printf

                strcpy

*/

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

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

        if (t->Type() == TIDENTIFIER)                   /* check for port variab
le */
        {
                strcpy(portName,t->Identifier());
                if (t->Next() == TINTEGER)              /* check for port addres
s */
                {    type = ADDRPORT;
                portAddress = t->Integer();
                        if ((((ii = t->Next()) == TKEYWORD) &&      /* direction
check */
                                  ((t->Keytype() == KINPUT) ||
                                   (t->Keytype() == KOUTPUT)))
                        {
                                direction = t->Keytype();
                                if (t->Next() != TSEMICOLON)        /* end of sta
tement */
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num(), e
e->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)         // bau
d rate
                                        {
                                                baudrate = t->Integer();
```

```
                                                    {       startbit = t->Integer();
                                                            if (t->Next() == TINTEGE
R)   // number of stop bits for COM port
                                                            {       stopbit = t->Int
eger();
                                                                    if (t->Next() ==
 TINTEGER)                  // parity for COM port
                                                                    {       paritybi
t = t->Integer();
                                                                            if (t->N
ext() == 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
                                                                    {       printf("
\n%s", t->Thisline());
                                                                            printf("
Line %d %-80s", t->Line_num(), ee->readerror(_E_INTEGEREXPT));
                                                                            ee->chec
kerrors(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->Thisli
ne());
                                                            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_
ee->readerror(_W_PARAMETERERR));
```

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

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

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


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

}

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

/* NAME:        device_t::device_t
        TYPE:           Class constructor

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

   INPUT PARAMETERS:    token_t*
the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     translate_t::parseDevices
                                                table::search

        SERVICING FUNCTIONS:    token_t::Type
                                                        token_t::Next
                                                        token_t::Keytype
                        token_t::Thisline
                                                        coil_t::coil_t(t
oken_t*)
                                                        pulse_t::pulse_t
(token_t*)
                                                        sensor_t::sensor
_t(token_t*)
                                                        programmable_t::
programmable_t(token_t*)

                                                        error_t::readerr
ors
```

```
strcpy

                printf
*/

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

device_t::device_t(token_t *t) : stmt_t ("Device_t")
{
        if (t->Type() == TIDENTIFIER)                                   // devic
e 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++; // Incre
ment only if new type
                                coiltypes++;
                                break;
                        case KSENSOR:                           // create sensor device
                                devicePtr = new sensor_t(t);
                                strcpy(typename, "SENSOR");
                                if (sensortypes == 0) numtypes++; // Inc
rement only if new type
                                sensortypes++;
                                break;
                        case KPULSE:                            // create pulse device
                                devicePtr = new pulse_t(t);
                                strcpy(typename, "PULSE");
                                if (pulsetypes == 0) numtypes++; // Incr
ement only if new type
                                pulsetypes++;
                                break;
                        case KPROGRAMMABLE:  // create programmable device
                                devicePtr = new programmable_t(t);
                                strcpy(typename, "PROGRAMMABLE");
                                if (progtypes == 0) numtypes++; // Incre
ment 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 incorrec
t definition
                                                // in funciton table
                                printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), e
e->readerror(_E_INVALDEVTYP));
                                ee->checkerrors(ERROR_T);

                        }
                }
                else
                {       printf("\n%s", t->Thisline());
```

```
                        printf("%-80s", t->Line_num(), ee->readerror(_E_
DEVTYPEXPTD));
                        ee->checkerrors(ERROR_T);
        }
          }
}

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

/* NAME:        coil_t::coil_t
   TYPE:                Class constructor

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

   INPUT PARAMETERS:    token_t*                                        the curr
ent token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:  device_t::device_t

        SERVICING FUNCTIONS:    token_t::Next
                                                                token_t::Identif
ier
                                                                token_t::Integer
                                                                table::search(st
mt_t*, char*, int, int)

                                                                SYSTEM
strcpy

                printf
*/

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

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(), e
e->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }
```

```
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_IDENTIFE
XPT));
                ee->checkerrors(ERROR_T);
        }

}

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

/* NAME:        sensor_t::sensor_t
        TYPE:           Class constructor

        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.

   INPUT PARAMETERS:    token_t*
the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     device_t::device_t

        SERVICING FUNCTIONS:    token_t::Next
                                                                token_t::Identif
ier
                                                                token_t::Integer
                                                                table::search(st
mt_t*, char*, int, int)

                                                                SYSTEM
strcpy

                printf
*/

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

sensor_t::sensor_t(token_t* t) : device_t ("Sensor_t")
{
        if (t->Next() == TIDENTIFIER)
        {
```

```
                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(), e
e->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_IDENTIFE
XPT));
                ee->checkerrors(ERROR_T);
        }

}

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

/* NAME:        pulse_t::pulse_t
   TYPE:                Class constructor

        FUNCTION:    Creates a pulse object and 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.

   INPUT PARAMETERS:    token_t*
        the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     token_t::Next
                                                                token_t::Identif
ier
```

```
ter
                                                                table::search(st
mt_t*, char*, int, int)

                                                                SYSTEM
strcpy

                printf
*/

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

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(), e
e->readerror(_E_SEMICOLEXPT));
                                        ee->checkerrors(ERROR_T);
                                }

                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_IDENTIFE
XPT));
                ee->checkerrors(ERROR_T);
        }

}

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

```
/* NAME:          programmable_t::programmable_t
   TYPE:                  Class constructor

        FUNCTION:    Creates a programmable object and 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.

   INPUT PARAMETERS:     token_t*
                the current token

   OUTPUT PARAMETERS:    None

   RETURN VALUES:                          void

        SERVICED FUNCTIONS:      device_t::device_t

        SERVICING FUNCTION:      token_t::Next
                                                                    token_t::Identif
ier
                                                                    token_t::Integer

                                                                    SYSTEM
strcpy

                printf

                strlen

*/

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

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;
```

```
#ifdef  VERBOSE
                                printf("Device parsed successfully\n");
#endif
                                return;
                        }
                        else
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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);
        }

        }

}

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

/* NAME:        wait_t::wait_t
   TYPE:        Class constructor

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

   INPUT PARAMETERS:    token_t*
        the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     Not used at present

        SERVICING FUNCTIONS:    token_t::Next

                                                                SYSTEM
printf
*/

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

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

```
}

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

/* NAME:         procedure_t::procedure_t
   TYPE:                 Class constructor

        FUNCTION:    Creates a procedure object and parses an executeable statem
ent
        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.

   INPUT PARAMETERS:    token_t*                              the current token

   OUTPUT PARAMETERS:   None

        RETURN VALUES:                  void

        SERVICED FUNCTIONS:     translate_t::parseProcedure

        SERVICING FUNCTIONS: token_t::Type
                                                                  token_t::Identif
ier
                                                                  table::search(st
mt_t*, char*, int, int)
                                                                  device_t::get_de
vType
                                                                  device_t::getDev
ptr
                                                                  coil_t::parse(to
ken_t*, coil_t*, exec_code*)
                                                                  sensor_t::parse(
token_t*, sensor_t*, exec_code*)
                                                                  pulse_t::parse(t
oken_t*, pulse_t*, exec_code*)
                                                                  programmable_t::
parse(token_t*, programmable_t*, exec_code*)
                                                                  wait_t::parse(to
ken_t*, exec_code*)

*/

/* ********************************************************************** */
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 eliminated
                        switch (devptr->get_devType())
                        {
                                case KCOIL:
                                        ((coil_t*) (devptr->getDevptr()))->parse
(t, devptr, &codes);
```

```
                                        ((sensor_t*) (devptr->getDevptr()))->par
se(t, devptr, &codes);
                                        break;
                                case KPULSE:
                                        ((pulse_t*) (devptr->getDevptr()))->pars
e(t, devptr, &codes);
                                        break;
                                case KPROGRAMMABLE:
                                        ((programmable_t*) (devptr->getDevptr())
)->parse(t, devptr, &codes);
                                        break;
                                case KWAIT:
                                        ((wait_t*) (devptr->getDevptr()))->parse
(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_IDENTIFE
XPT));
                ee->checkerrors(ERROR_T);
        }

}

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

/* NAME:        coil_t::parse
   TYPE:                Member function

   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.

   INPUT PARAMETERS:    token_t*
        the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     procedure_t::procedure_t

        SERVICING FUNCTIONS:    token_t::Next
                                                        token_t::Keytype
                                                        table::search(st
mt_t*, char*, int, int)
                                                        device_t::get_de
vType
                                                        coil_t::get_port
Name
                                                        port_t::get+port
Address
```

```
*/

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

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)   // should
be 5
                                {
                                        for (int j=0;!valid && j<MAXFUNC;j++)
                                        {
                                                // search through the function t
able
                                                if ((t->Keytype() == j+FUNCINDEX
) &&
                                                        (valid = functbl[i][j].v
alid))
                                                {
                                                        code->opcode = functbl[i
][j].opcode;
                                                        code->address = ((port_t
*)((sytb->search(NULL,
                                                        ((coil_t* )(dt->getDevpt
r()))->get_portName(),
                                                        KPORTS,0))->symbol))->ge
t_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->reade
rror(_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->reade
rror(_E_SEMICOLEXPT));
                                ee->checkerrors(ERROR_T);
                        }

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

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

}

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

/* NAME:        sensor_t::parse
   TYPE:        Member function

   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.

   INPUT PARAMETERS:    token_t*
                the current token

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     procedure_t::procedure_t

        SERVICING FUNCTIONS:    token_t::Next
                                                                token_t::Keytype
                                                                table::search(st
mt_t*, char*, int, int)
                                                                device_t::get_de
vType
                                                                sensor_t::get_po
rtName
                                                                port_t::get+port
Address

                                                                SYSTEM
printf
*/

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

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+FUNCINDEX
) &&
                                                                (valid = functbl[i][j].v
alid))
```

```
                                                        {
                                                                code->opcode = functbl[i
][j].opcode;
                                                                code->address = ((port_t
*)((sytb->search(NULL,
                                                                ((sensor_t* )(dt->getDe\
ptr()))->get_portName(),
                                                                KPORTS,0))->symbol))->ge
t_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->reade
rror(_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->reade
rror(_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_STMTNOEF
FEC));
                        ee->checkerrors(WARNING_T);
                }

}

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

/* NAME:        pulse_t::parse
        TYPE:           Member function

   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.

        INPUT PARAMETERS:       token_t*        t
                the current token
```

```
   RETURN VALUES:                          void

         SERVICED FUNCTIONS:      procedure_t::procedure_t

         SERVICING FUNCTIONS:     token_t::Next
                                                                  token_t::Keytype
                                                                  table::search(st
mt_t*, char*, int, int)
                                                                  device_t::get_de
vType
                                                                  pulse_t::get_por
tName
                                                                  port_t::get+port
Address

                                                                  SYSTEM
printf

*/

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

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+FUNCINDEX
) &&
                                                        (valid = functbl[i][j].v
alid))
                                                {
                                                        code->opcode = functbl[i
][j].opcode;
                                                        code->address = ((port_t
*)((sytb->search(NULL,
                                                        ((pulse_t* )(dt->getDevp
tr()))->get_portName(),
                                                        KPORTS,0))->symbol))->ge
t_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->reade
rror(_E_INVALIDFUNC));
                                ee->checkerrors(ERROR_T);
                        }
                        if (t->Next() == TSEMICOLON)
          {
#ifdef  VERBOSE
```

```
error();
                                        return;
                                }
                                else
                                {       printf("\n%s", t->Thisline());
                                        printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_STMTNOEF
FEC));
                        ee->checkerrors(WARNING_T);
                }

}

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

/* NAME:        programmable_t::parse
        TYPE:           Member function

        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.

        INPUT PARAMETERS:       token_t*

   OUTPUT PARAMETERS:   None

   RETURN VALUES:                       void

        SERVICED FUNCTIONS:     procedure_t::procedure_t

        SERVICING FUNCTIONS:    token_t::Next
                                                                token_t::Keytype
                                                                token_t::Identif
ier
                                                                token_t::Integer
                        token_t::Type
                                                                table::search(st
mt_t*, char*, int, int)
                                                                device_t::get_de
vType
                                                                device_t::getDev
ptr
                                                                programmable_t::
get_comportName

                                                                SYSTEM
printf

                strlen
```

```
   CALLED FUNCTIONS:
*/

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

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+FUNCINDEX
) &&
                                                        (valid = functbl[i][j].v
alid))
                                {
                                                        code->opcode = functbl[i
][j].opcode;
                                                        switch (porttype)
                                                        {
                                                                case SERPORT:
                                                                        pp = syt
b->convertPort((sytb->search(NULL,
        comportName, KPORTS, 0))->symbol);
                                                                        code->co
mport = new char[strlen(pp->Typename())+1];
                                                                        strcpy(c
ode->comport, pp->Typename());
                                                                        break;
                                                                case PARPORT:
                                                                        code->co
mport = new char[strlen(comportName)+1];
                                                                        strcpy(c
ode->comport, comportName);
                                                                        break;
                                                                default:
                                                                        break;
                                                        }
                                                }
                                        }  // end of 2nd for
                                        if (!valid)
                                        {       printf("\n%s", t->Thisline());
                                                printf("Line %d %-80s", t->Line_
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) &&
```

```
                                                    {
                                                            if (t->Type() == TCOMMA
!! t->Type() == TQUOTE)
                                                                    t->Next();
                                                            if (functbl[i][j-1].plis
t[k] == t->Type())
                                                            {
                                                                    if (t->Type() ==
 TCHARSTRING)
                                                                    {
                                                                            code->st
ring = new char[strlen(t->Identifier())+1];
                                                                            strcpy(c
ode->string, t->Identifier());
                                                                    }
                                                                    if (t->Type() ==
 TIDENTIFIER)
                                                                    {
                                                                            code->fi
lename = new char[strlen(t->Identifier())+1];
                                                                            strcpy(c
ode->filename, t->Identifier());
                                                                    }
                                                                    validparam = !va
lidparam;
                                                                    k++;
                                                            }
                                                            else
                                                            {       validparam = CIN
VALID;
                                                                    printf("\n%s", t
->Thisline());
                                                                    printf("Line %d
%-80s", t->Line_num(), ee->readerror(_E_PARTYPMISMA));
                                                                    ee->checkerrors(
ERROR_T);
                                                            }

                                                    } // END OF WHILE
                                                    if (t->Next() == TSEMICOLON)
                        {
#ifdef  VERBOSE
                                                            printf("Procedure statem
ent parsed successfully\n");
#endif
                                                            return;
                                                    }
                                                    else
                                                    {       printf("\n%s", t->Thisli
ne());
                                                            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->Line_
num(), ee->readerror(_E_SEMICOLEXPT));
                                                    ee->checkerrors(ERROR_T);
                                            }
```

```
                        } // end of 1st for
                        if (!valid)
                        {       printf("\n%s", t->Thisline());
                                printf("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_STMTNOEF
FEC));
                ee->checkerrors(WARNING_T);
        }

}


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

/* NAME:        wait_t::parse
        TYPE:           Member function

        FUNCTION:    Parses a wait device type declaration according to the synt
ax
        defined in the CPL grammar.  If any error is discovered, an appropriate
    error message is printed out.

        INPUT PARAMETERS:       token_t*

    OUTPUT PARAMETERS:   None

    RETURN VALUES:                          void

        SERVICED FUNCTIONS:     procedure_t::procedure_t

        SERVICING FUNCTIONS:    token_t::Next
                                                                token_t::Integer

                                                                SYSTEM
printf
    CALLED FUNCTIONS:
*/

/* ********************************************************************** */
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("Line %d %-80s", t->Line_num(), ee->reade
rror(_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_STMTNOEF
FEC));
                        ee->checkerrors(WARNING_T);
                }

}

/* end of file */
```

```
//-------------------------------------[ 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                                 /* Strin
g */
#define SQTE                                4                           /* Quote */
#define SLNF                                5                           /* Linefeed stat
e */
#define SDONE             10                /* final state */
```

```
short action_table[][] = {
            /*  States
                                                S  S   S   S   S   S
                                                I  I   I   T   Q   L
                                                N  D   N   R   T   N
                                                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
};


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

/* NAME:          token_t::Next
   TYPE:                  member function

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

        INPUT PARAMETERS:        void

   OUTPUT PARAMETERS:    void

   RETURN VALUES:                          int      // indicating token type

        SERVICED FUNCTIONS:  translate_t::parsePorts
                                                                translate_t::par
seDevices
                                                                translate_t::par
seProcedure
                                                                port_t::port_t
                                                                device_t::device
_t
                          procedure_t::Procedure_t
                                                                port_t::parse
                                                                device_t::parse
                                                                coil_t::coil_t
                                                                sensor_t::sensor
_t
                                                                pulse_t::pulse_t
                                                                programmable_t::
programmable_t
                                                                wait_t::wait_t
                                                                coil_t::parse
                                                                sensor_t::parse
                                                                pulse_t::parse
                                                                programmable_t::
parse
                                                                wait_t::parse

        SERVICING FUNCTIONS:     Char::next

                                                                Char::Class
                                                                Char::code
                                                                Char::reset
```

```
cout

                sizeof

                strcmp
*/

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


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 */
```

```
                        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(keyw
ords[0]); i++)
                        {
                                if (strcmp(string,keywords[i]) == 0)
                                {
                                        token = TKEYWORD;
                                        keytype = i;
                                }
                        }
                        // Check for keyword so that keyword lines are n
ot printed into
        // the trace file
                        if ((token != TKEYWORD) && CheckSend() && (newli
ne))
                                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;
```

```
                                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 */
```

```
//-----------------------------------[ 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 */
};

/*  ********************************************************************************** */
/* NAME:         char::Class
        TYPE:               member function

        FUNCTION:           Reads one source code line from the cpl program file

   INPUT PARAMETERS:        None
   OUTPUT PARAMETERS:       None

        RETURN VALUES:                      void

        CLIENT FUNCTIONS:                   token_t::Next

        SERVICE FUNCTIONS:    error_t::readerror
                                                              error_t::checker
rors
```

```
        strcpy

        printf

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

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);
        }

}
/*  ************************************************************************ */


/*  ************************************************************************ */
/* NAME:         char::Next
   TYPE:                  member function

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

   INPUT PARAMETERS:     None
   OUTPUT PARAMETERS:    None

   RETURN VALUES:                          int

   CLIENT FUNCTIONS:     token_t::Next

        SERVICE FUNCTIONS:

*/

/*  ************************************************************************ */
int char_t::next(void)
{
        if (reget_flag)
        {
                reget_flag = 0;     // Reset reget flag if set
        } else
        {
                if (nextline[nextchar] == '*')      // check for comment characte
r
      {
                        last_char = '\n';
                        nextchar = 0;
                }
                else
                {       last_char = nextline[nextchar];
                        nextchar++;
      }
        }
        return(last_char);     // Return previous character if reget flag set
}
/*  ************************************************************************ */

/* NAME:         char::Class
   TYPE:                  inline function
```

```
er.

   INPUT PARAMETERS:     None
   OUTPUT PARAMETERS:    None

        RETURN VALUES:                    short

        CLIENT FUNCTIONS:                 token_t::Next

   SERVICE FUNCTIONS:    None

*/

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

// 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_clas
s[last_char] : CEOF); }


/* end of file */
```

```
//-----------------------------------[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

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

/* NAME:         translate_t::parsePorts
         TYPE:              member function

   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.

         INPUT PARAMETERS:        void

   OUTPUT PARAMETERS:    None

   RETURN VALUES:                          void

         SERVICED FUNCITONS:      main

         SERCICING FUNCTIONS: table::insert(stmt_t*, char*, int)
                                                                  port_t::port_t(t
oken_t*)
                                                                  token_t::Next
                                                                  token_t::Keytype
                                                                  token_t::Line_nu
m
```

```
feof

                printf

*/

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

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_PORTKEYEX
PT));
                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);
}

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

/* NAME:        translate_t::parseDevices
   TYPE:        member function

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

        INPUT PARAMETERS:       void

   OUTPUT PARAMETERS:   None

        RETURN VALUES:                  void

        SERVICED FUNCITONS:     main

        SERCICING FUNCTIONS: table::insert(stmt_t*, char*, int)
                                                        device_t::device
_t(token_t*)

                                                        token_t::Next
                                                        token_t::Keytype
                                                        token_t::Line_nu
m



feof
```

```
                printf

*/

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

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_DEVKEYWEX
PT));
                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);

}

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

/* NAME:        translate_t::parseProcedure
   TYPE:        Class constructor

   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.

        INPUT PARAMETERS:       void

   OUTPUT PARAMETERS:   None

        RETURN VALUES:                  int

        SERVICED FUNCITONS:     main

        SERCICING FUNCTIONS: table::insert(stmt_t*, char*, int)
                                                                procedure_t::pro
cedure_t(token_t*)
                                                                token_t::Next
                                                                token_t::Keytype
                                                                token_t::Line_nu
m

                                                                SYSTEM
feof

                printf

*/
```

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

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 PROCE
DURE
                        (t->Keytype() == KPROCEDURE));
                if (!result)       // if not generate error
                {
                        printf("%s", t->Thisline());
                        printf("Line %d %s\n", t->Line_num(), ee->readerror(_E_P
ROCKEYEXPT));
                        ee->checkerrors(ERROR_T);
                }

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

}

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

/*  NAME:         translate_t::generate
    TYPE:         member function

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

   INPUT PARAMETERS:     void

   OUTPUT PARAMETERS:    None

        RETURN VALUES:           int

        SERVICED FUNCITONS:      main

        SERVICING FUNCTIONS: list_t::first
                                                              list_t::next

                                                              SYSTEM
        fopen

                        fclose

                        feof
```

```
                        fgets

                        strcpy

                        strcat

                        fprintf

                        printf

*/

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

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 symbol
table
        port_t* pt;                                        // port type pointer

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

        code = procPtr->first();
   (code->getdevicepts() --> generate();
   // 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->get_
direction())// Set up direciton string
                                                                {       case KIN
PUT: strcpy(directionStr,"INPUT"); break;
                                                                        case KOU
TPUT: strcpy(directionStr,"OUTPUT"); break;
                                                                        default:
 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->Star
tBit(), 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->code
s.opcode, code->codes.address, code->codes.bit, SETBIT);
                                        break;
                                case COFF:
                                        fprintf(fp, "%d %ld %d %d\n", code->code
s.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                        break;
                                case CWAITON:
                                        fprintf(fp, "%d %ld %d %d\n", code->code
s.opcode, code->codes.address, code->codes.bit, SETBIT);
                                        break;
                                case CWAITOFF:
                                        fprintf(fp, "%d %ld %d %d\n", code->code
s.opcode, code->codes.address, code->codes.bit, RESETBIT);
                                        break;
                                case CSEND:
                                        fprintf(fp, "%d %s %s\n", code->codes.op
code, code->codes.comport, code->codes.string );
                                        break;
                                case CWAIT:
                                        fprintf(fp, "%d %d\n", code->codes.opcod
e, code->codes.waitsecs);
                                        break;
                                case CSTROBE:
                                        fprintf(fp, "%d %ld %d %d\n", code->code
s.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, CMDEXT),
 "r")) != NULL)
                                        {
                                                while (!feof(rp))
                                                {
                                                        if (fgets(sendstr, MAXST
```

```
                                                        fprintf(fp, "%d %s %s",
code->codes.opcode, code->codes.comport, sendstr );
                                                }
                                                fclose(rp);
                                        }
                                        else
                                        {       printf("%s %s\n", ee->readerror(
_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);
}

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

/*  NAME:       translate_t::crossrefer
    TYPE:       member function

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

        INPUT PARAMETERS:    void

   OUTPUT PARAMETERS:   None

        RETURN VALUES:                  int

        CALLED FUNCTIONS:    cc                 build                    int int

        print                   char* int int

        display         char*

        CALLING FUNCITONS:      main
*/

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

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.display(refile);             // display cross references
        return (1);
}
```

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

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


error_t::error_t(char* fname)
{
        if ((ep = fopen(fname, "r")) == NULL)
    {
                printf("%s\n", readerror(_F_UNOPNERRFIL));
                checkerrors(FATAL_T);
        }
        else
    {
                err_count = 0;
                warn_count = 0;
                tot_count = 0;
                err_limit = 5;
        }
}

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_UNREADERRFIL));
```

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

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", to
t_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", to
t_count);
					exit(-1);
				}
				break;
			case FATAL_T:
				printf("Fatal error discovered\n");
				printf("Compilation cannot continue\n");
				printf("%d errors discovered during compilation\n", tot_
count);
				exit(-2);
				break;
			default:
				break;
		}
}
```