Graphical User Interface Environment for Developing Workcell Control Programs

Zhili Jin
Miami University, commons-admin@lib.muohio.edu

TECHNICAL REPORT: MU-SEAS-CSA-1995-004

Graphical User Interface Environment for Developing Workcell Control Programs

Zhili (Jake) Jin

Graphical User Interface Environment for Developing Workcell Control Programs

by

Zhili (Jake) Jin
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #95-004 July, 1995

Graphical User Interface Environment for Developing Workcell Control Programs

July, 1995

Author: Jake Zhili Jin

Advisor: Mr. Douglas A. Troy

Department of Systems Analysis

Miami University

Oxford, OH 45056

Graphical User Interface Environment for Developing Workcell Control Programs

Abstract

A forms-based development environment for writing manufacturing workcell programs is described. The environment is a Microsoft Windows application that allows a programmer to describe a sequence of operations and associated preconditions to control a workcell by filling in forms. The environment then generates code to be loaded into a controller. Currently, the environment generates a language called Cell Programming Language (CPL), which is a workcell programming language used at Miami University. The aim of the environment is to provide an easier-to-use environment for developing workcell control software.

Acknowledgments

It's been fun and exciting exploring many fine features of Microsoft C++ and creating the manufacturing worcell program. I want to thank the Department of Systems Analysis and the Department of Manufacturing Engineering for providing a very good working environment and lab facilities for me to implement this project. In particular, I appreciate Mr. Troy's guidance and suggestions on the target of this project, and exhaustive reviews for improvements of this paper. Also Zhuming Wang gave many good points to help me to understand the operation of manufacturing system in the Engineering Lab.

1. Introduction

A flexible manufacturing system (FMS) is a manufacturing system that is reprogrammable and capable of producing a variety of products automatically. It can be considered as a set of workcells that operate and are scheduled independently of each other (Chang, 87, and Benhabib, 89). A workcell is composed of one or more machine tools linked by a common material handling system and under the control of a centralized workcell controller for the purpose of producing the given requirements of a family of parts. The workcell controller is programmed to coordinate the interoperation of the various devices in the workcell. The major advantages of FMSs are high machine utilization, flexibility in production scheduling, and high labor productivity (Martin, 89, and Groover, 80).

Meghamala (1992) developed an Object-Oriented High-Level Language called Cell Programming Language (CPL) for programming an individual workcell controller. CPL allows a user to program a workcell by referring to devices as objects (robot and conveyor, for example) and using commands such as On/Off to control the devices, and hides the low-level programming details. Wang (1994) further extended this language to support flow control, error recovery, and operator interface.

Computer Aided Software Engineering (CASE) refers to the application of software to assist in some aspect of the software development process. The term 'software engineering' has become increasingly used in all types of software development. Software used for such tasks as editing, compiling, test administration and so on are referred to as 'software tools' (McDermid, 1991). The purpose of this project is to develop a Windows programming tool as an aid for users to more easily develop CPL code. The programming environment will replace the use of a simple text editor with Windows dialogs and will then generate CPL code automatically. Furthermore, the user will define the program as a series of steps, which mirror the manufacturing process to be performed. In this way not only does a user understand the logic of each step, it is easier to develop a CPL program and there are fewer typing errors during development stage. This is very important for engineering students who have limited experience in programming.

The remainder of this report will describe the new programming environment. Section 2 presents an example of a flexible manufacturing workcell and describe the existing CPL workcell programming system. Section 3 describes the design of the new programming environment, and Section 4 shows its implementation. Section 5 is a discussion of issues encountered during the implementation of the new environment. Section 6 summarizes the results of the project, and Section 7 suggests future work related to it.

2. A Flexible Manufacturing System and the CPL Coding Environment:

The CIM lab in the Manufacturing Engineering Department at Miami University is a typical example of an FMS workcell (Figure 1). The flexible manufacturing cell consists of an Emco Maier Compact-5 CNC lathe, a Span Tech XL Loop conveyor system, RM-501 Mitsubishi robot, pallet stops, a pallet lift, and various sensors. A relay is used to activate the conveyor, solenoids are used for the lathe chuck, the pallet stops, and the pallet lift. The conveyor motor is powered by 110 VAC, while the pallet stops, lift, and chuck are powered pneumatically. Sensors include a photocell used to detect the approach of a pallet to the machining station on the conveyor, a switch to detect the arrival of pallet to the station, and a switch to determine if the pallet has been lifted up to the robot's pick-up position. The inputs and outputs of these devices are wired, through external relay interfacing, to a data acquisition board in an IBM compatible PC. The robot controller is connected to the PC's printer port, which is used to send the robot movement commands in immediate mode. The CNC lathe is connected to a serial port on the PC, which is used to program the lathe. The robot and CNC machine are programmed using their native language.

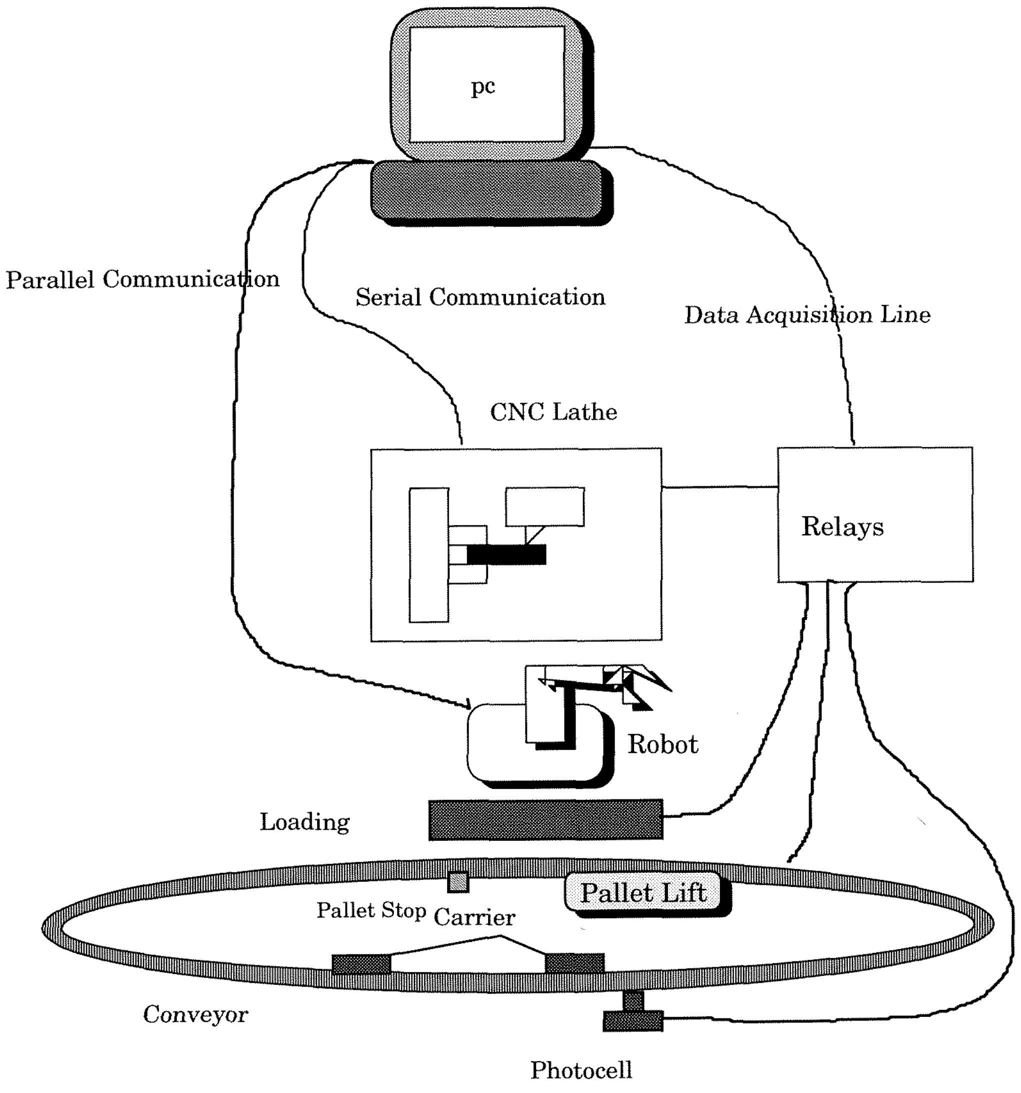


Figure 1. The Flexible Manufacturing System in CIM Lab, Miami University

Cell Programming Language (CPL) is an object-like workcell programming language developed at Miami University for senior undergraduate engineering students to program the PC to control the workcell. CPL consists of three parts: the CPL language, the compiler, and the interpreter. The language allows a user to describe the sequence of the operations required to manufacture a part in the FMS workcell. The compiler then translates the CPL source code into intermediate code called p-code. Finally the interpreter takes p-code as input and executes it to control the operation of the devices. Figure 2 shows the CPL software architecture (Troy, 1992).

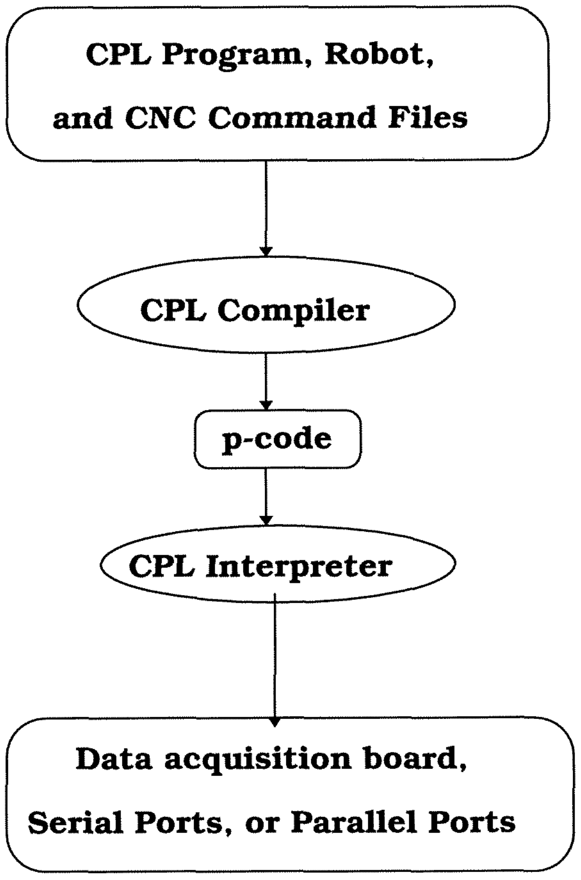

CPL Program, Robot, and CNC Command Files
CPL Compiler
p-code
CPL Interpreter
Data acquisition board, Serial Ports, or Parallel Ports

Figure 2. CPL Software Architecture (Troy, 1992)

A CPL program consists three major sections: port declarations, device declarations, and procedure statements (Wang, 1994). The port declaration section is used to assign a physical port address on the data acquisition board in the PC. The declarations are made within a *Ports ... End* block. Following the keyword *Ports* is a series of individual port declarations. Its syntax is as follows:

<port_name> <port_address> <direction>

The *port_name* can be any user defined identifier consisting of a maximum of 30 characters. The *port_address* is a physical port address and the *direction* is either Input or Output depending on whether the port is used to receive or send signals. An example of port declaration section is given bellow:

```
Ports
        PortA     642        Output;
        PortB     643        Input;
        PortC     644        Output;
End
```

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. The declaration block is bounded by the keywords *Devices* and *End.* The syntax is as follows:

<device_name> <device_type> <port_name> [<bit_num>]

The *device_name* is a user-defined identifier and the *device_type* is a keyword in the language. The *port_name* is defined earlier in the port declaration section and the *bit_num* is a constant between 0 and 7 and corresponds to a bit within a byte on the data acquisition board. For programmable devices, the port name LPT1 or COM1 is specified instead of port variable and bit. An example of device declaration section looks like:

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

The last 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 in the cell. The syntax of a procedure statement is as follows:

<device_name>.<device_function>[(param{,param ...})]

The *device_name* is an identifier previously declared in the device declaration section. The *device_function* is predefined and a keyword in the language. Table 1 lists device types and valid functions for each device type.

DEVICE TYPE	**FUNCTION**
Coil	On, Off
Sensor	WaitOn, WaitOff
Pulse	Strobe
Programmable	Send, Do
Wait	Time

Table 1. Device types and their functions

Function parameters are enclosed within parenthesis and are separated by commas. Similar to the devices and ports section, the keywords *Procedure* and *End* mark the beginning and end of the procedure block. Following is an example:

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

With CPL, engineering students can program device objects with names that directly correspond to their real-world counterparts. The predefined device functions are named after the actual device operations. For example, a statement such as Conveyor.On is an instruction to switch on the conveyor and a statement such as PhotoCell.WaitOn is an instruction to wait for the photocell to be switched on. This way it is possible to write a program and visualize the cell's operation.

To create a CPL program, students need to type their whole procedure source code using a text editor, for example the DOS Editor. There are several disadvantages with this environment: (1) Typing errors are common. The compiler is case sensitive and repeated typing of the same port and device names can cause definition errors. (2) The editor does not check for proper functions for each device, so error checking is deferred until compile time. Different types of devices have different functions, for example, the robot is a programmable type device, and has Send and Do functions, as shown in Table 1. (3) The editor does not provide help facilities for creating CPL programs. (4) The environment is not integrated -- the editor, compiler, and interpreter are all separate tools.

The purpose of this project is to develop a Windows environment and overcome the above disadvantages. Under the new environment, students only need to define devices and specify the corresponding types once. The device types are controlled by program and allow selection from a list. The procedures are defined through a series of dialogs. In this way, students can focus on their logical design, instead of typing, and the resulting code is exactly what they want. Once the students finish the design phase, they can generate CPL source code and save it as a file which can then be compiled.

Another goal of the new environment is that it will allow the workcell programmer to organize the operation of the workcell into logical steps. A step will consist of a number of preconditions, followed by one or more device operations. This way, the new environment provides a higher level of abstraction in comparison to CPL. This will be further explained in Section 3.

3. Design of the New Environment:

A complete CPL program can be seen to be a sequence of manufacturing steps. For example, consider the CPL procedure shown in Table 2. Notice that most steps (except steps 1 and 5) have an initial precondition that must be satisfied, followed by one or more operations. For example, the precondition for step 3 is that the PalletArrived condition be true, after which the Pallet will be lifted. Thus the procedure section consists of one or more steps where each step consists of zero or more precondition statements and at least one device operation.

CPL Procedure	**Logical Steps**

LatheG66inp.Strobe;
Delay.500;
Lathe.Do(loadpart);
Robot.Send("NT");
PalletStops.On;
Conveyor.On;
	Step 1: Initialize

PhotoCell.WaitOn(500);
PalletStops.Off;
	Step 2: Wait for Part

PalletArrived.WaitOn(500);
PalletLiftUp.Strobe;
Delay.1000
	Step 3: Lift Part

PalletLifted.WaitOn(500);
Conveyor.Off;
ChuckOpen.Strobe;
Robot.Do(loadpart);
Delay.1000;
ChuckClose.Strobe;
Delay.2000;
Robot.Do(moveaway);
Delay.2000;
LatheStart.Strobe;
	Step 4: Manufacture the Part

Robot.Do(moveback);
Delay.2000;
ChuckOpen.Strobe;
Delay.2000;
Conveyor.On;
Delay.500;
Conveyor.Off;
PalletStops.Off;
LatheStart.Strobe;
LatheHandShk.Strobe;
	Step 5: Unload the Part

Table 2: A typical Procedure Section of a CPL Program and its steps.

In the new environment, a user will group statements into steps. For a step definition, the user is only allowed to select different devices and the corresponding actions from lists and the new environment will combine device, action, and parameters, and generate CPL statements automatically. Microsoft Visual C++ is used to implement the Windows environment.

Eight dialog windows are created to handle user interface. The corresponding C++ dialog classes are shown in Table 3, and their hierarchical organization is shown in Fig. 3. With the new environment, a programmer must define ports, devices, and procedures as in CPL, but instead of using an editor, he or she will fill in forms using Windows dialogs. These dialogs are described further in Section 4.

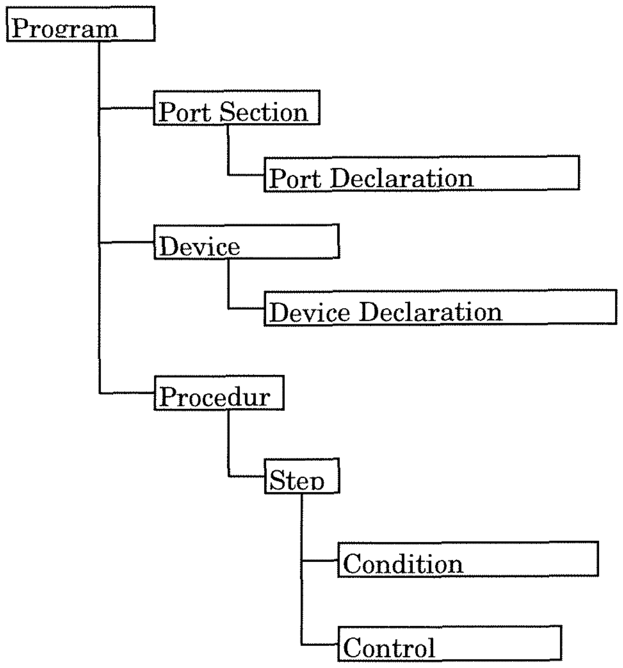


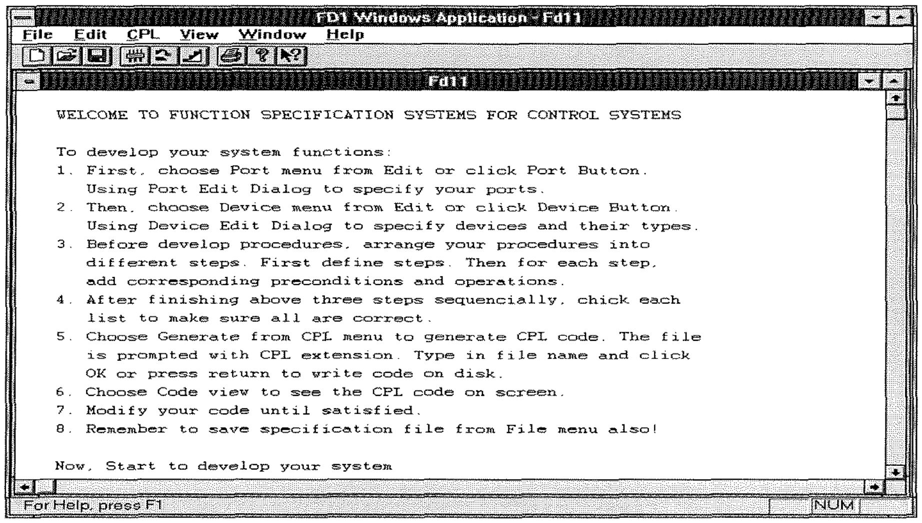
Fig. 3. The Hierarchy of Statement Classes

Dialog Class	**Purpose**
CPort	to define and edit port name, address, and I/O type.
CPortList	to list defined ports.
CDeviceEdit	to define and edit device name, and to specify its type and port used
CDeviceList	to list devices defined.
COperateEdit	to select device, action, and to specify delay time.
CPreconditionEdit	to select device, its condition, and wait time.
CStepEdit	to define a logical step and to list its preconditions and operations.
CStepList	to list logical steps defined in CStepEdit Dialog.

Table 3: Eight Dialog Classes used for Specifying CPL Program

4. Implementation of the New Environment

The top level dialog for the new environment is shown in Fig. 4. The window also shows instructions how to use the program. In the following are the steps to specify a CPL program and generate the code using the new environment.

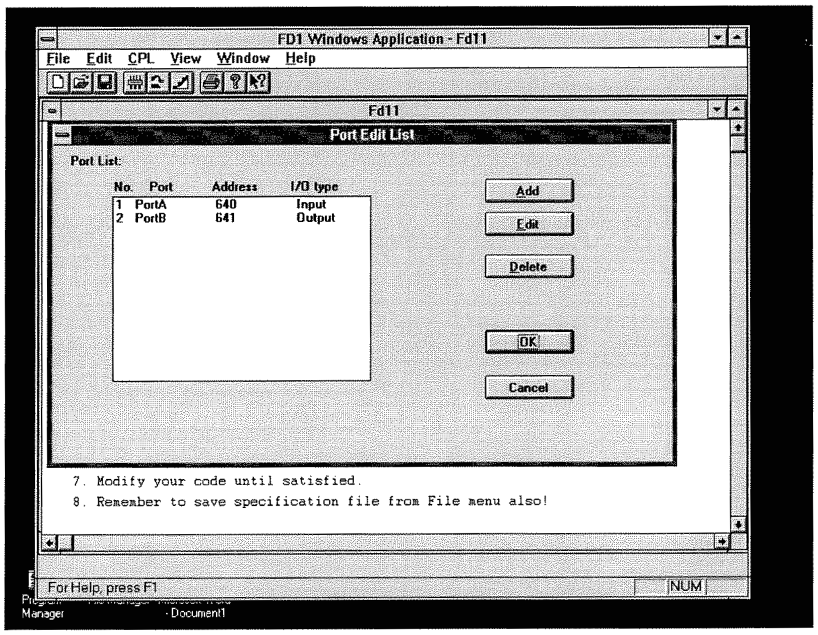

FD1 Windows Application - Fd11
File Edit CPL View Window Help
Fd11
For Help, press F1
NUM

```
WELCOME TO FUNCTION SPECIFICATION SYSTEMS FOR CONTROL SYSTEMS

To develop your system functions:
1. First, choose Port menu from Edit or click Port Button.
   Using Port Edit Dialog to specify your ports.
2. Then, choose Device menu from Edit or click Device Button.
   Using Device Edit Dialog to specify devices and their types.
3. Before develop procedures, arrange your procedures into
   different steps. First define steps. Then for each step,
   add corresponding preconditions and operations.
4. After finishing above three steps sequencially, chick each
   list to make sure all are correct.
5. Choose Generate from CPL menu to generate CPL code. The file
   is prompted with CPL extension. Type in file name and click
   OK or press return to write code on disk.
6. Choose Code view to see the CPL code on screen.
7. Modify your code until satisfied.
8. Remember to save specification file from File menu also!

Now, Start to develop your system
```

Fig. 4. The entry screen for designing and generating CPL code

Step 1. Port Definition

The user can either select Port under Edit menu or simply click the Port button on the toolbar -- the fourth button from the left. Then the Port List dialog appears on screen. If this is the first time to edit ports, the port list is blank. Figure 5 shows port list dialog after adding two ports. To add a port, click the Add button. Then the Port Edit dialog window appears, as shown in Figure 6. The user types in the port name and its address, and selects the I/O type. The port number will be increased automatically. Once the user clicks the OK button, the port item will be inserted into port list. The user can also edit existing port definitions by selecting an item in the list, clicking the Edit button, or simply double clicking the item. Once finishing the port definition, the user can click the OK button to go back to the main window. If the user clicks the Cancel button, no new definitions will be entered.

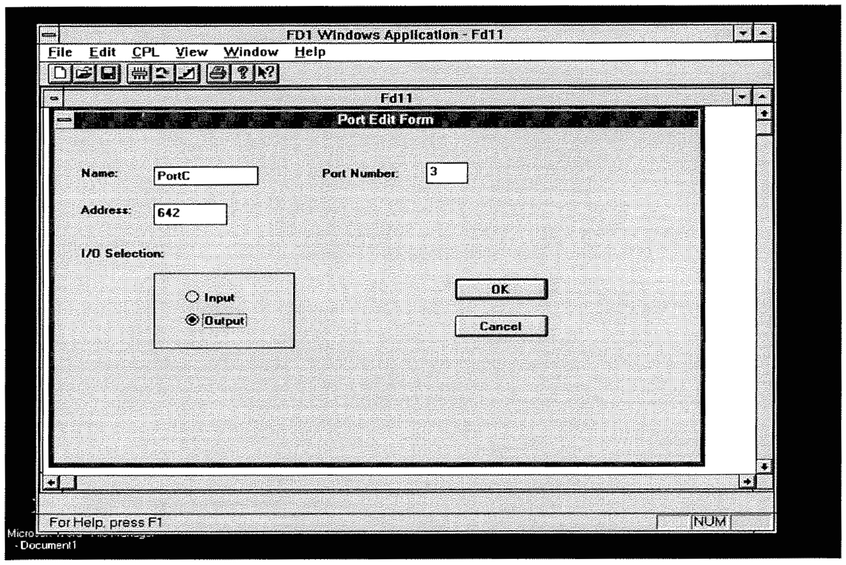

FD1 Windows Application - Fd11
File Edit CPL View Window Help
Fd11
Port Edit List
Add
Edit
Delete
OK
Cancel
For Help, press F1
NUM

Port List:

No.	Port	Address	I/O type
1	PortA	640	Input
2	PortB	641	Output

```
7. Modify your code until satisfied.
8. Remember to save specification file from File menu also!
```

Fig. 5. Port List Screen showing that two ports are defined


FD1 Windows Application - Fd11
File Edit CPL View Window Help
Fd11
Port Edit Form
OK
Cancel
For Help, press F1
NUM

Name: PortC

Port Number: 3

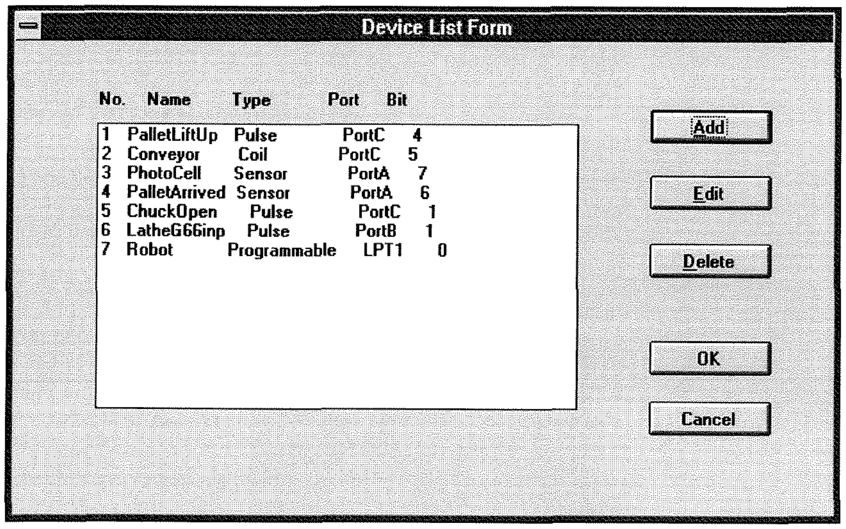
Address: 642

I/O Selection: Input [✓ Output]

Fig. 6. Port Edit screen showing that PortC editing

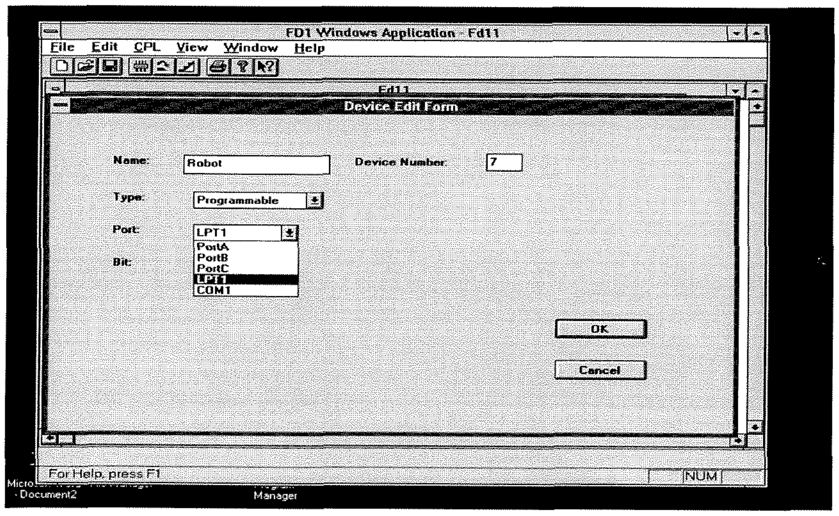
Step 2. Device Definition

Similar to port definition, device list and edit dialog windows are used to define the devices. Figure 7 shows the device list dialog. The dialog is activated by clicking the fifth toolbar button (Device toolbar), or selecting Device under the Edit menu to define devices. It lists device names and types, the required port name, and the bit number. For the device edit dialog, the user only needs to type in a device name and then and select its type and the port from lists in the dialog. The device types are predefined, and the ports are those defined earlier plus LPT1 and COM1, as shown in Fig. 8. By using selection instead of typing, there is no possibility of typing errors for this process.



No.	Name	Type	Port	Bit
1	PalletLiftUp	Pulse	PortC	4
2	Conveyor	Coil	PortC	5
3	PhotoCell	Sensor	PortA	7
4	PalletArrived	Sensor	PortA	6
5	ChuckOpen	Pulse	PortC	1
6	LatheG66inp	Pulse	PortB	1
7	Robot	Programmable	LPT1	0

Fig. 7. The Device List Screen listing device name, type, required port name, and bit number defined.

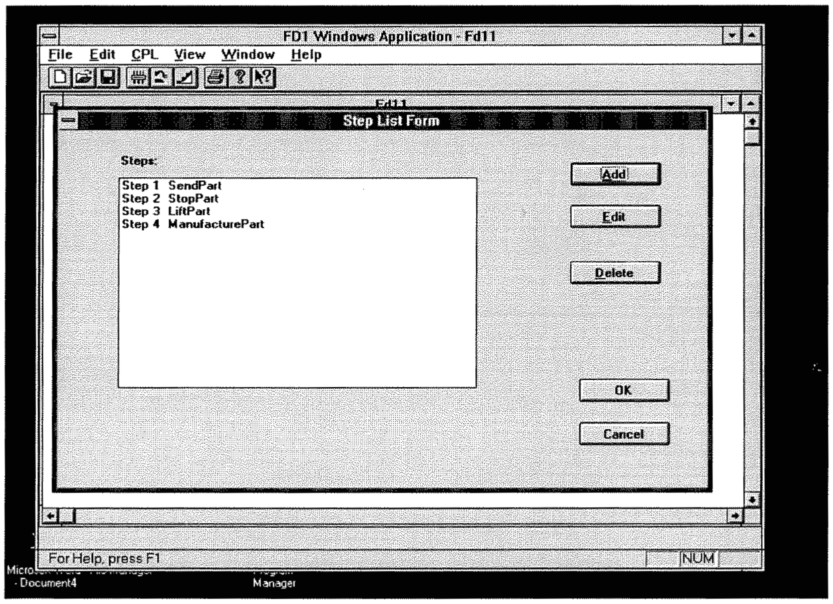


Name: Robot
Device Number: 7
Type: Programmable
Port: LPT1
Bit:

Fig. 8. Device Edit Screen allows a user to define a device name, and select its type and required port name.

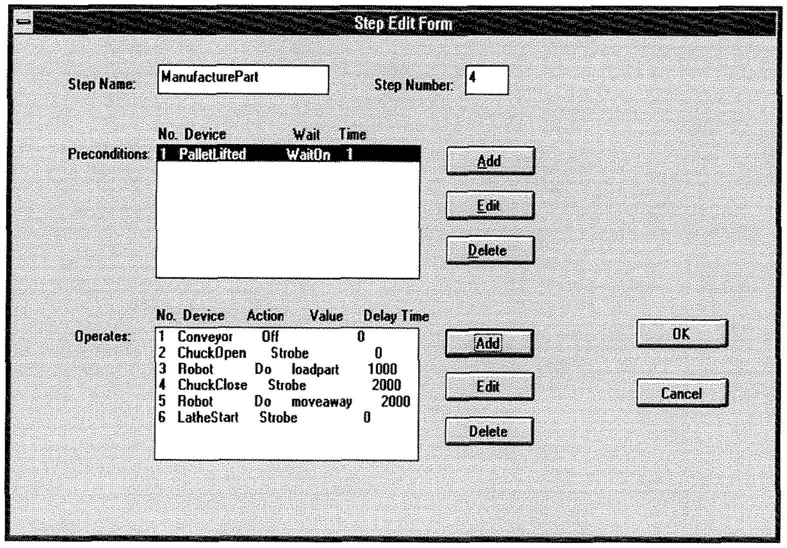
Step 3. Step Definition

Like the port and device list dialogs, a step list dialog window is used to create steps by clicking the sixth toolbar or select Step under Edit menu to edit steps. Fig. 9 shows the screen after adding four steps. To add a step, click the Add button and go to the Step Edit dialog window, as shown in Fig. 10.



```
Step 1  SendPart
Step 2  StopPart
Step 3  LiftPart
Step 4  ManufacturePart
```

Fig. 9. The Step List Screen showing that four steps are entered.



Step Edit Form

Step Name: ManufacturePart   Step Number: 4

Preconditions:

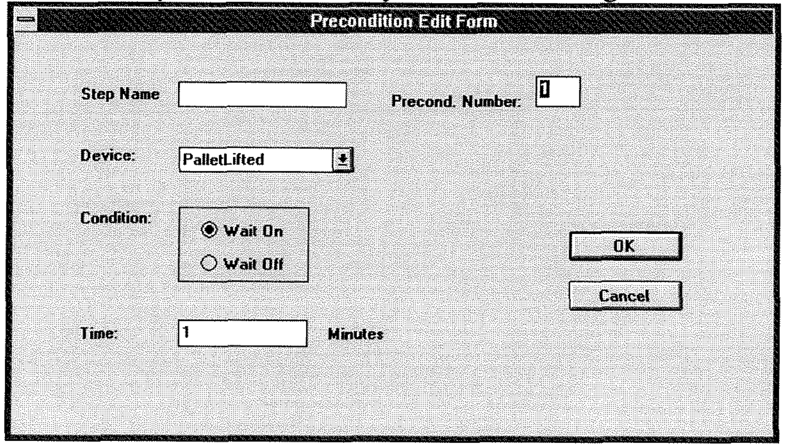
No.	Device	Wait	Time
1	PalletLifted	WaitOn	1

Operates:

No.	Device	Action	Value	Delay Time
1	Conveyor	Off		0
2	ChuckOpen	Strobe		0
3	Robot	Do	loadpart	1000
4	ChuckClose	Strobe		2000
5	Robot	Do	moveaway	2000
6	LatheStart	Strobe		0

Fig. 10. The Step Edit Screen allows a user to define a step name, and add/edit a precondition or operation.

The user needs to type in a meaningful step name, like *ManufacturePart*, and then add preconditions and operations by clicking the corresponding Add buttons. For a precondition, a device defined earlier is selected from the list in a combo box, then select the condition (Wait On/Off), and specify a maximum waiting time, as shown in Fig. 11. For an operation, once the user selects a device, its available actions, which are determined by the device type, are listed a list box. The programmer selects a desired action, and specifies a delay time if necessary, as shown in Fig. 12.



Precondition Edit Form

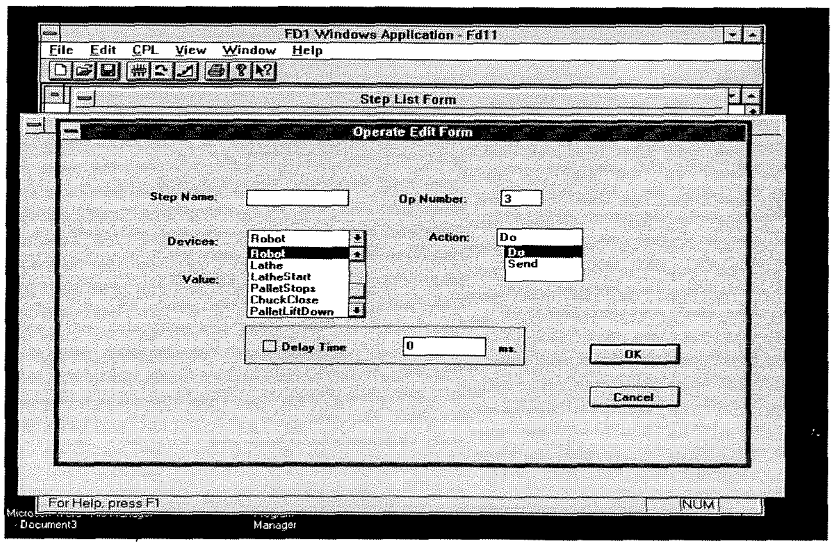
Step Name: Precond. Number: 0

Device: PalletLifted

Condition: [✓ Wait On] Wait Off

Time: 1 Minutes

Fig. 11. The Precondition Edit Screen allows a user to select a defined device from the list box and specify waiting condition.


FD1 Windows Application - Fd11
File Edit CPL View Window Help
Step List Form
Operate Edit Form
OK
Cancel
For Help, press F1
NUM

Step Name:

Op Number: 3

Devices: Robot

Robot
Lathe
LatheStart
PalletStops
ChuckClose
PalletLiftDown

Value:

Action: Do

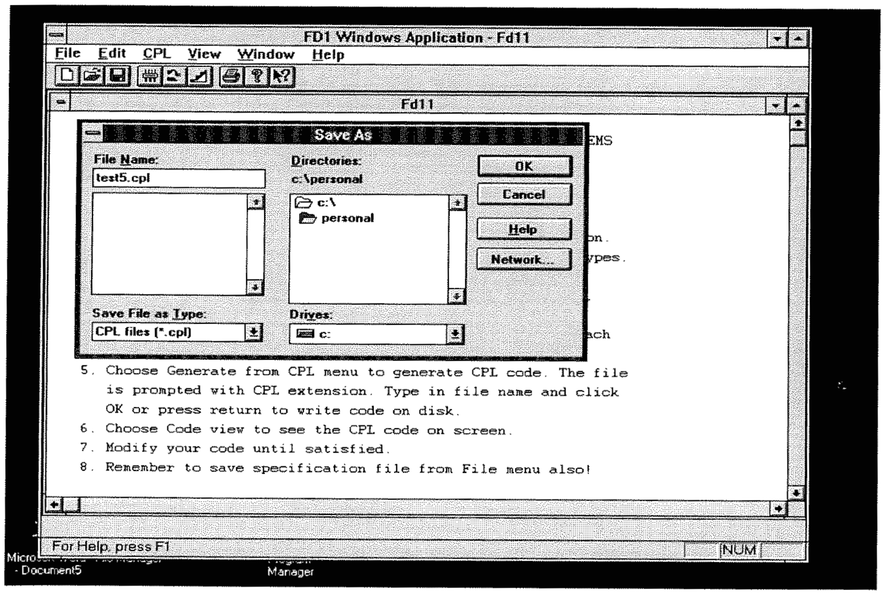
Do
Send

☐ Delay Time 0 ms.

Fig. 12. The Operation Edit Screen allows a user to select a device from device box and its action list on the right box, and specify delay time if necessary.

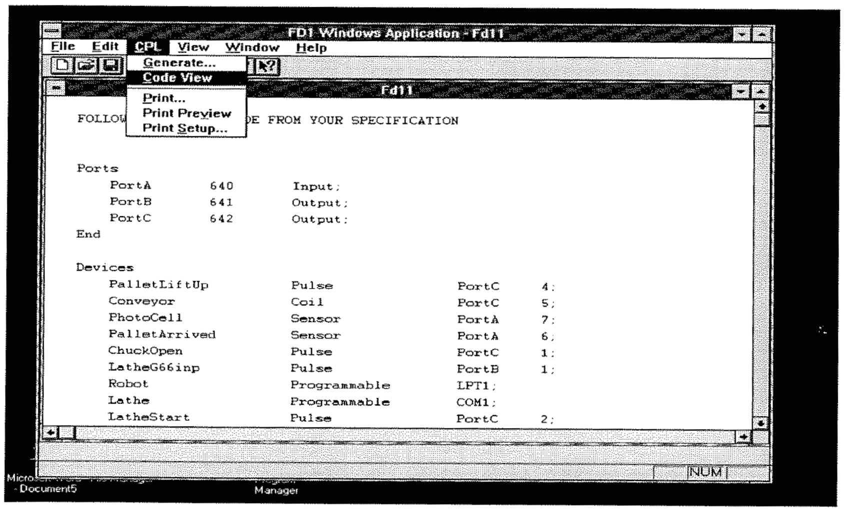
Step 4. CPL Code Generation

After finishing step definition, the user is ready to generate CPL source code. To do that, the programmer selects Generate under the CPL menu. A File Save dialog will appear as shown in Fig. 13 to allow the programmer to type in a file name for the CPL code (the default extension is .cpl). After clicking the OK button, the source code is generated and is saved in the file. To view the CPL code, select Code View under the CPL menu, and the source code is listed on screen as shown in Fig. 14. The user also can print the code out by selecting Print under the CPL menu. Fig. 15 shows a printout of CPL code.



```
5. Choose Generate from CPL menu to generate CPL code. The file
   is prompted with CPL extension. Type in file name and click
   OK or press return to write code on disk.
6. Choose Code view to see the CPL code on screen.
7. Modify your code until satisfied.
8. Remember to save specification file from File menu also!
```

Fig. 13. The File Save Dialog allows a user to save CPL source code as a disk file. The default extension is CPL.



```
FOLLOW                E FROM YOUR SPECIFICATION


Ports
    PortA       640       Input;
    PortB       641       Output;
    PortC       642       Output;
End

Devices
    PalletLiftUp          Pulse            PortC     4;
    Conveyor              Coil             PortC     5;
    PhotoCell             Sensor           PortA     7;
    PalletArrived         Sensor           PortA     6;
    ChuckOpen             Pulse            PortC     1;
    LatheG66inp           Pulse            PortB     1;
    Robot                 Programmable     LPT1;
    Lathe                 Programmable     COM1;
    LatheStart            Pulse            PortC     2;
```

Fig. 14. The Entry Screen displays CPL code when a user selects Code View under CPL menu.

```
Ports
    PortC       642       Output;
    PortA       640       Input;
    PortB       641       Output;
End

Devices
    PalletLiftUp          Pulse              PortC     4;
    Conveyor              Coil               PortC     5;
    PhotoCell             Sensor             PortA     7;
    PalletArrived         Sensor             PortA     6;
    ChuckOpen             Pulse              PortC     1;
    LatheG66inp           Pulse              PortB     1;
    Robot                 Programmable       LPT1;
    Lathe                 Programmable       COM1;
    LatheStart            Pulse              PortC     2;
    LatheStop             Sensor             PortA     4;
    PalletLifted          Sensor             PortA     5;
    PalletStops           Coil               PortC     0;
    ChuckClose            Pulse              PortC     3;
    PalletLiftDown        Pulse              PortC     6;
    LatheRunning          Sensor             PortA     2;
    LatheHandShk          Pulse              PortB     0;
End

Procedure
//Step_One Precondition list:

//Step_One Operation list:
    LatheG66inp.Strobe;
    Delay.500
    Lathe.Do(loadlathe);
    Robot.Send("NT");
    PalletStops.On;
    Conveyor.On;

//Step_Two Precondition list:
    PhotoCell.WaitOn(5);

//Step_Two Operation list:
    PalletStops.Off;

//Step_Three Precondition list:
    PalletArrived.WaitOn(5);

//Step_Three Operation list:
    PalletLiftUp.Strobe;
    Delay.1000

//Step_Four Precondition list:
    PalletLifted.WaitOn(5);

//Step_Four Operation list:
    Conveyor.Off;
    ChuckOpen.Strobe;
    Robot.Do(loadpart);
    Delay.1000
    ChuckClose.Strobe;
    Delay.2000
    Robot.Do(moveaway);
    Delay.2000
    LatheStart.Strobe;

//Step_Five Precondition list:
    LatheStop.WaitOn(5);

//Step_Five Operation list:
    Robot.Do(moveback);
    Delay.2000
    ChuckOpen.Strobe;
    Delay.2000
    Robot.Do(getpart);
    PalletStops.On;
    PalletLiftDown.Strobe;
    Conveyor.On;
    Delay.500
    Conveyor.Off;
    PalletStops.Off;
    LatheStart.Strobe;
    LatheHandShk.Strobe;

End
```

Fig. 15 CPL Code Generated by the New Programming Environment

5. Discussion of Using Microsoft Visual C++

Microsoft Visual C++ contains a very powerful Windows-based application framework -- the framework on which programmers build applications for Windows. At a general level, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. It is an integrated collection of object-oriented software components that offers all that's needed for a generic application and is a superset of a class library (Kruglinski, 1993, p18). It defines the structure of the program itself. Microsoft Foundation Class (MFC) Library version 2.0 is an important part of Visual C++ and the core of the application framework. The MFC Library consists of a library of C++ classes and global functions with source code included. Other components -- including AppWizard, ClassWizard, AppStudio, Visual Workbench, the compiler, and the linker-- are the tools used to construct applications. Some of the classes encapsulate a large portion of the Microsoft Windows application programming interface (API). Other classes encapsulate application concepts such as documents, views, and the application itself.

AppWizard is a code generator that creates a working skeleton of a Windows application with features, class names, and source code filenames. Its purpose is to get a programmer started quickly with a new application. AppWizard creates all of the necessary files and classes for the application type.

App Studio includes both a *WYSIWYG* menu editor and a powerful dialog box editor. Use App Studio to design an application's user interface and create the application's resources: menu, dialog boxes, custom controls, accelerator keys, bitmaps, icons, cursors, and strings.

ClassWizard is a program that operates both inside the Visual Workbench and inside App Studio. ClassWizard takes the drudgery out of maintaining Visual C++ class code. ClassWizard writes the prototypes, function bodies, and code to connect the messages to the application framework. The ClassWizard sets up the "message map" structure necessary for connecting the application framework to your function's code.

Following are a summary of the sequence of steps in building an application with Microsoft Visual C++:

1. Run AppWizard to create the files for a skeleton application, including source files for your application, document, view, and frame windows, a resource file, and a project file (.MAK).
2. Use AppStudio to visually edit the application's user interface, including menus, accelerators, dialog boxes, bitmaps, icons, cursors, and other resources.
3. Use ClassWizard to connect menus and accelerators to handler functions, insert message-map entries and empty function templates in the source files.
4. Use Visual Workbench editor to fill in the code for your handler functions.
5. Create additional classes if necessary. ClassWizard adds these classes to source files and helps programmers define their connections to any commands they handle.
6. Implement application-specific document class(es). Add member variables to hold data structures. Add member functions to provide an interface to the data. The framework already knows how to interact with document data files. It can open and close document files, read and write the document's data, and handle other user interfaces.
7. Implement Open, Save, and Save AS commands by writing code for the document's Serialize member function. The framework displays dialog boxes for the Open, Save, and Save As commands on the File menu. It writes and reads back a document using the data format specified in your Serialize member function.

8. Implement one or more view classes corresponding to documents. Implement the view's member functions that will be mapped to the user interface with ClassWizard. The framework manages most of the relationship between a document and its view. The view's member functions access the view's document to render its image on the screen or printed page and to update the document's data structures in response to user editing commands.
9. Enhance other features, like printing, print preview, scrolling, splitter windows, if necessary.
10. Use the facilities of Visual Workbench to build, test, and debug the application. Visual Workbench is closely coupled with AppWizard, App Studio, and ClassWizard. It lets programmer adjust compile, link, and other options. And it lets them browse their code and class structure.

6. Conclusion

The following are the results of this project:

1. The new working environment provides a visual and straight forward environment for developing CPL code. It completely overcomes typing errors from text editor environment.
2. During CPL code development, students design their logical procedures on the computer using forms and the concepts of steps.
3. Device types and actions are given as prompts (lists) by the program, so for each type of device, only legal actions can be selected. This is a form of help for the student programmers and helps eliminate typing errors.
4. Visual C++ provides very powerful tools for programmers to develop Windows application quickly. It fully supports object-oriented programming.
5. A weakness of the new environment is that the CPL compiler and interpreter are not integrated into the environment. After generating CPL code, the programmer must compile and execute it outside of the Windows environment.

Table 4 below summarized the new environment in comparison to the old .

Features	**New**	**Old**
GUI	Yes	No
Typing Errors	Eliminated	No Control
Checking Device Functions	Yes	No
Help	Yes	No
View Control Using Steps	Yes	No
Integrated Compiler and Execution	No	No

Table 4. Comparison of New and Old Environments

7. Future Work

The new environment can be improved in several ways, as listed below.

1. Currently, the program can only generate CPL code. It should have the ability to run the CPL compiler and interpreter to execute the CPL code.
2. Wang (1994) extended the CPL to include flow control and conditional execution. Correspondingly, the program needs to be updated to coordinate with this capability.
3. Additional rules could be added. For example, the environment could check that the same bit is not used more than once for a specific port.
4. This program is mainly used for the workcell of CIM Lab. It is possible to extend this program to more general manufacturing systems.
5. Context-sensitive help can be added. Most Windows-based programs take advantage of the powerful WINHELP help engine that is included with Windows. The MFC Library version 2.0 application framework allows programmers to use this same help engine for context-sensitive help in applications. Manufacturing knowledge and terminology can be added for students reference.

References

Benhabib, B., C. Y. Chen, and W. R. Johnson**, An Integrated Manufacturing Work Cell Management System**, *Manufacturing Review*, Vol. 2, No. 4, 1989.

Chang, T. C., R. A. Wysk, H. P. Wang, **Computer-Aided Manufacturing**, *Prentice-Hall, Inc.,* Englewood Cliffs, N.J., 1987.

Groover, M. P., Automation, **Production Systems, and Computer Integrated Manufacturing**, *Prentice-Hall, Inc.*, Englewood Cliffs, N.J., 1980.

Kruglinski, D. J., **Inside Visual C++,** *Microsoft Press*, 1993, p 7-13

Martin, J. M., **Cells Drive Manufacturing Strategy**, *Manufacturing Engineering*, Jan., p.49-54, 1989.

McDermid, J. A., **Software Engineer's Reference Book**, *Butterworth Heinemann*, p 33/3 -17, 1991

Meghamala, N., **Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler**, *Working Paper #92-015*, Dec. 1992.

Troy, D. A., M. Nugehally, S. Farooq, D. Hergert, **Object-Oriented Flexible Manufacturing System at Miami University**, *Proceedings of ICOOMS'92*, May 1992.

Wang, Zhuming, **Cell Programming Language Investigation of Extensions for Flow Control and Error Recovery in the Language**, *Working Paper #94-005*, 1994.

HEALTHSTAR, Inc.

7257 North Lincoln Avenue
Lincolnwood, Illinois 60646
708 673-3113
Fax 708 679-4804

send application

Jake Zhili Jin
2702 S. Union Ave.
Chicago, IL 60616
(312) 842-6609

March 30, 1995

Dear Mr. Troy:

Included are a copy of my updated paper and the disk. Fig. 4 to Fig. 15 are in zip files. I hope that you have some time to look it. So I can make any changes necessary and finish it before the end of this semester. Thank you!

Jake

HealthStar

Offices in: Illinois, Ohio, Georgia, Indiana, Kentucky, Alabama, Iowa, Nebraska