Consequently, the robot controller software works as a server, exposing to the client a collection of RPC services that constitute its basic functionality. Those services, offered by the RPC servers running on the robot controller, include the variable access services, files and programs management services, and robot status and controller-state management and information services. To access those services, the remote computer (*client*) issues properly parameterized remote procedure calls to the robot controller (*server*) through the network.

Considering, for example, the S4CPLUS robot controller from ABB Robotics, it's possible to extend the RPC services available in the robot controller adding user functionality to the system. The ABB implementation is based on a messaging protocol developed by ABB called RAP (*remote application protocol*) [8], which is an *application specific protocol* (ASP) of the OSI application level. The messaging protocol RAP defines the necessary data structures and message syntax of the RPC calls used to explore the RPC services available in the controller.

These services were implemented using the standard and open source RPC specification SUN RPC 4.0, a collection of tools developed by the *SUN Microsystems Open Network Group* (ONC) [2]. Consequently, to implement the client calls, the *ONC SUN RPC 4.0* specification and tools were also used. This package includes a compiler (*rpcgen*), a *portmaper* and a few useful tools like *rpcinfo*. The Microsoft RPC implementation uses another standard defined by *Digital Corporation* named *OSF/DCE*, which is not compatible with the SUN RPC standard. The package used to build the client software was a port to *Windows NT/2000/XP*, equivalent to the original version that was built to *UNIX* systems, although a few functions were slightly changed to better suit the needs without compromising compatibility with client and server programs developed with the *SUN RPC* package. The port was compiled using the *Microsoft Visual C++ .NET 2003* compiler.

From all the RPC services available in the robot controller, the ones really needed to implement the software architecture depicted in Figure 3.10 are the variable access services. Nevertheless, calls to the other services were implemented for completeness. The procedure is simple and based on the XDR (*extended data representation*) file obtained by defining the data structures, the service identification numbers, and the service syntax specified by the RAP protocol. That file is compiled by the *rpcgen* tool, generating the basic calls and data structure prototypes necessary to invoke the RPC services available from the robot controller. The necessary code was added to each basic function and packed into an *ActiveX* software component named *PCROBNET2003/5* [5-7]. The complete set of functions included in this object is listed in Table 3.3.

Although this software component was built using the DCOM/OLE/ActiveX object model, it does not run the *Microsoft RPC* implementation but instead the already mentioned *SUN RPC 4.0* port to the *Win32* API.

Table 3.3 Methods and properties of the software component PCROB NET2003/5

Function	Brief description
open	Opens a communication line with a robot (RPC client)
close	Closes a communication line
motor_on	Go to run state
motor_off	Go to standby state
prog_stop	Stop running program
prog_run	Start loaded program
prog_load	Load named program
prog_del	Delete loaded program
prog_set_mode	Set program mode
prog_get_mode	Read actual program mode
prog_prep	Prepare program to run (program counter to begin)
pgmstate	Get program controller state
ctlstate	Get controller state
oprstate	Get operational state
sysstate	Get system state
ctlvers	Get controller version
ctlid	Get controller ID
robpos	Get current robot position
read_xxxx	Read variable of type xxxx (there are calls for each type of variable defined in RAPID)
read_xdata	Read user-defined variables
write_xxx	Write variable of type xxxx (there are calls for each type of variable defined in RAPID)
write_xdata	Write user-defined variables
digin	Read digital input
digout	Set digital output
anain	Read analog input
anaout	Set analog output

To use a remote service, the computer running the client application needs to make properly parameterized calls to the server computer, and receive the execution result. Two types of services may be considered: synchronous and asynchronous. The synchronous services return the execution result as the answer to the call.

Consequently, the general prototype of this type of call is:

short *status call_service_i (**struct** parameters_i, **struct** answer_i)*

where *status* returns the service error codes (zero if the service returns without errors, and a negative number identifying the error otherwise), *parameters_i* is the data structure containing the service parameters and *answer_i* is the data structure that returns the service execution results.

The asynchronous services, when activated, return answers/results asynchronously, *i.e.*, the remote system should also make remote procedure calls to the client system when the requested information becomes available or when the specified event occurs (system and controller state changes, robot program execution state change, IO and variable events, *etc.*). Those calls may be named events or spontaneous messages, and are remote procedure calls issued to all client computers that made the correspondent subscription, *i.e.*, made a call to the subscription service properly parameterized specifying the information wanted. To receive those calls, any remote client must run RPC servers that implement a service to receive them (Figure 3.13). The option adopted was to have that server broadcast registered messages inside the operating system, enabling all open applications to receive and process that information by filtering its message queue.

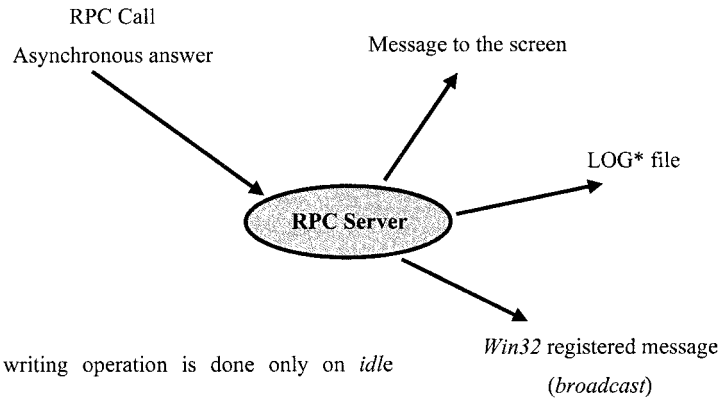


Figure 3.13 Functionality of the RPC server necessary to receive spontaneous messages

As mentioned already, the variable access services allow access to all types of variables defined in the robot controller. Using this service, and developing the robot controller software in a convenient way, it is possible to add new services to the system. In fact, that possibility may be achieved by using a simple SWITCH-CASE-DO cycle driven by a variable controlled from the calling (client) remote computer:

```
switch (decision_1)
{
    case 0: call service_0; break;
    case 1: call service_1; break;
    case 2: call service_2; break;
    ...
    case n: call service_n; break;
}
```

This server runs on the robot controller, making the process of adding a new service a simple task. The programmer should build the procedure (routine) that implements the new functionality, and include the call to that procedure in the server cycle by identifying it with the specific number of the control variable.

This is not far from what is done with any RPC server; the *svc_run* function, used in those programs is a *SWITCH-CASE-DO* cycle that implements calls to the functions requested by the remote client. With this type of structure it is straightforward to build complex and customer functions that can be offered to the remote client. Furthermore, with this approach it's still possible to use the advanced capabilities offered by the robot controller, namely the advanced functions designed to control and setup the robot motion and operation. Examples exploring this facility are presented and discussed in this chapter (sections 3.4 to 3.6).

3.3.2 TCP/IP Sockets

One of the most interesting ways to establish a network connection between computer systems is by using TCP/IP sockets. This is a standard client-server procedure, not dependent on the operating system technology used on any of the computer systems, requiring only the definition of a proper messaging syntax to be reliable and safe. The user-defined messaging protocol should specify the commands and data structures adapted to the practical situation under study.

The TCP/IP protocol suite is based on a four-layer reference model. All protocols that belong to the TCP/IP protocol suite are located in the top three layers of this model.

As shown in Figure 3.14, each layer of the TCP/IP model corresponds to one or more layers of the seven-layer *Open Systems Interconnection* (OSI) reference model proposed by the *International Standards Organization* (ISO).

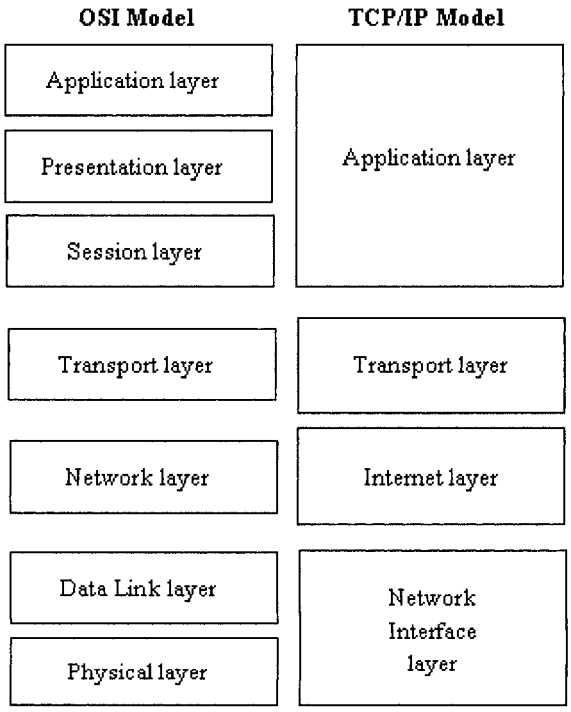


Figure 3.14 Correspondence between the OSI Model and the TCP/IP Model in terms of layers.

Table 3.4 Services performed at each layer of the TCP/IP Model

Layer	**Description**
Application	Defines the TCP/IP application protocols and how the host programs interface with transport layer services to use the network
Transport	Provides communication session management between host computers. Defines the level of service and the status of the connection used when transporting data
Internet	Packages data into IP datagrams, which contain source and destination address information that is used to forward the datagrams between hosts and across networks. Performs routing of IP datagrams
Network interface	Specifies details of how data is physically sent through the network, including how bits are electrically signaled by hardware devices that interface directly with a network medium, such as coaxial cable, optical fiber, or twisted-pair copper wire

The types of services performed at each layer within the TCP/IP model are described in more detail in Table 3.4.

Transmission control protocol (TCP) is a required TCP/IP standard defined in RFC 793, "*Transmission Control Protocol (TCP)*" that provides a reliable, connection-oriented packet delivery service. The *transmission control protocol*:

- Guarantees delivery of IP datagrams
- Performs segmentation and reassembly of large blocks of data sent by programs
- Ensures proper sequencing and ordered delivery of segmented data
- Performs checks on the integrity of transmitted data by using checksum calculations
- Sends positive messages depending on whether data was received successfully. By using selective acknowledgments, negative acknowledgments for data not received are also sent
- Offers a preferred method of transport for programs that must use reliable session-based data transmission, such as client/server database and e-mail programs

TCP is based on point-to-point communication between two network hosts. TCP receives data from programs and processes this data as a stream of bytes. Bytes are grouped into segments that TCP then numbers and sequences for delivery.

Before two TCP hosts can exchange data, they must first establish a session with each other. A TCP session is initialized through a process known as a *three-way handshake*. This process synchronizes sequence numbers and provides control information that is needed to establish a virtual connection between both hosts.

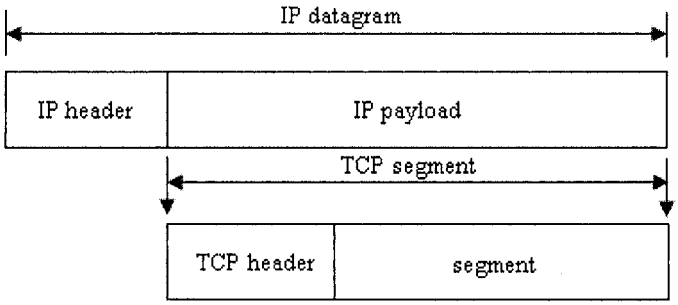


Figure 3.15 TCP segment within an IP datagram

Once the initial *three-way handshake* completes, segments are sent and acknowledged in a sequential manner between both the sending and receiving hosts. A similar handshake process is used by TCP before closing a connection to verify that both hosts are finished sending and receiving all data.

TCP segments are encapsulated and sent within IP datagrams, as shown in Figure 3.15

3.3.2.1 TCP Ports
TCP ports use a specific program port for delivery of data sent by using the *transmission controlpProtocol.* TCP ports are more complex and operate differently from UDP ports.

While a UDP port operates as a single message queue and the network endpoint for UDP-based communication, the final endpoint for all TCP communication is a unique connection. Each TCP connection is uniquely identified by dual endpoints. Each single TCP server port is capable of offering shared access to multiple connections because all TCP connections are uniquely identified by two pairs of IP address and TCP ports (one address/port pairing for each connected host).

The server side of each program that uses TCP ports listens for messages arriving on their well-known port number. All TCP server port numbers less than 1024 (and some higher numbers) are reserved and registered by *the Internet Assigned Numbers Authority* (IANA).

3.3.3 UDP Datagrams

The *User Datagram Protocol* (UDP) is a TCP/IP standard defined in RFC 768, "*User Datagram Protocol (UDP)*". UDP is used by some programs instead of TCP for fast, lightweight, unreliable transportation of data between TCP/IP hosts.

UDP provides a connectionless datagram service that offers best-effort delivery, which means that UDP does not guarantee delivery or verify sequencing for any datagrams. A source host that needs reliable communication must use either TCP or a program that provides its own sequencing and acknowledgment services.

UDP messages are encapsulated and sent within IP datagrams, as shown in 3.16.

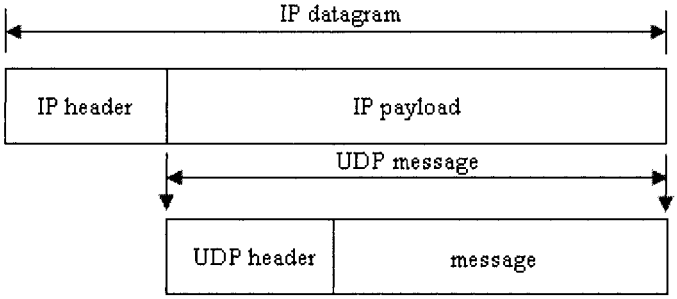


Figure 3.16 UDP message within an IP datagram

3.3.3.1 UDP Ports
UDP ports provide a location for sending and receiving UDP messages. A UDP port functions as a single message queue for receiving all datagrams intended for the program specified by each protocol port number. This means UDP-based programs can receive more than one message at a time.

The server side of each program that uses UDP listens for messages arriving on their well-known port number. All UDP server port numbers less than 1024 (and some higher numbers) are reserved and registered by the Internet *Assigned Numbers Authority* (IANA).

Each UDP server port is identified by a reserved or well-known port number.

3.4 Simple Example: Interfacing a CCD Camera

The example presented in this section demonstrates the utilization of TCP/IP sockets (stream type or TCP sockets) to command an industrial robot and to interface with a CCD camera (a common USB Webcam). The example will be presented in detail with the objective of allowing the reader to explore further from the concepts and ideas presented.

Basically the system is composed of the following components (Figure 3.17):

- Industrial robot manipulator ABB IRB140 equipped with the new IRC5 robot controller
- Pneumatic tool equipped with a vacuum cup
- Working table and several working pieces
- Webcam used to obtain the number of pieces present in the scene and their respective positions
- Pocket PC running the *Windows Mobile 2005* operating system

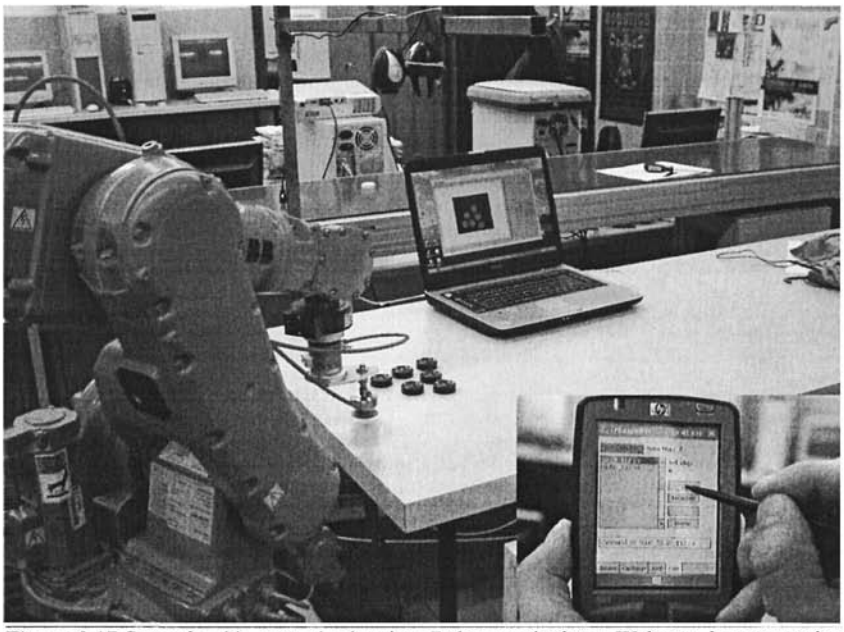


Figure 3.17 Setup for this example showing: Robot manipulator, Webcam, laptop running the Webcam TCP/IP server, and the commanding Pocket PC

The user is supposed to control the setup using the Pocket PC, namely:

- Change the robot state and start/stop program execution
- Interface with the Webcam, request the camera to identify the number of objects present in the scene and return their actual positions (Figure 3.18)
- Command the robot to pick-and-place the selected objects

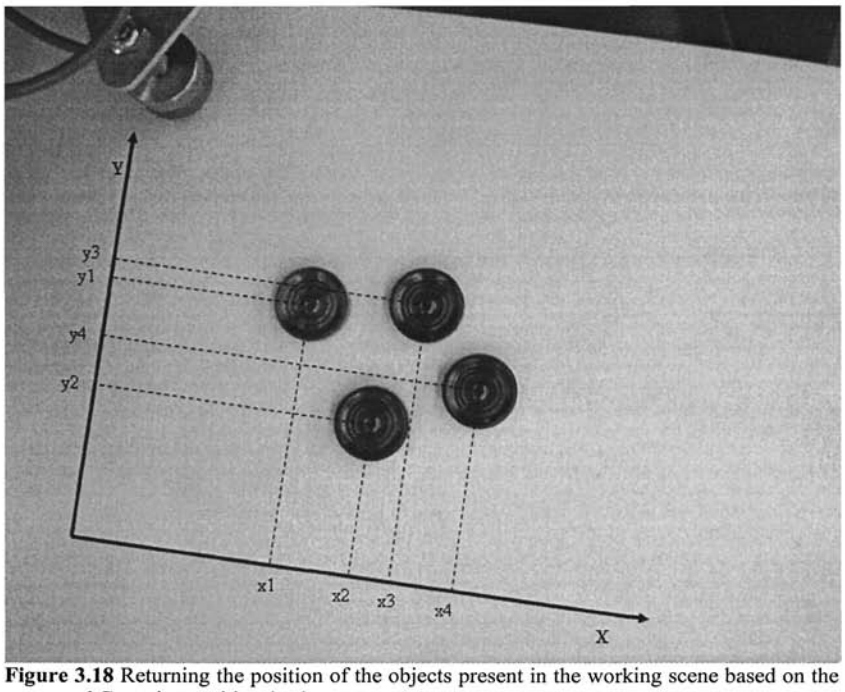


Figure 3.18 Returning the position of the objects present in the working scene based on the computed Cartesian position (x,y)

To build a solution to execute the above specified functions, it is necessary to handle several different subjects:

- Build a TCP/IP socket server to run on the robot controller. The server should implement a collection of services equivalent to the ones listed in Table 3.3
- Build an application to handle the webcam permitting to use it as a sensor and return the number of objects in the scene and their position. That application must run on a machine accessible from the network
- Build an application to command the setup offering a *human-machine interface* (HMI) facility

The following section provides a closer look at these three software packages.

3.4.1 Robot Controller Software

The robot controller runs two very different types of applications:

- The socket server used to implement the remote services and offer them to the remote clients
- The application that executes the commanded pick-and-place operations

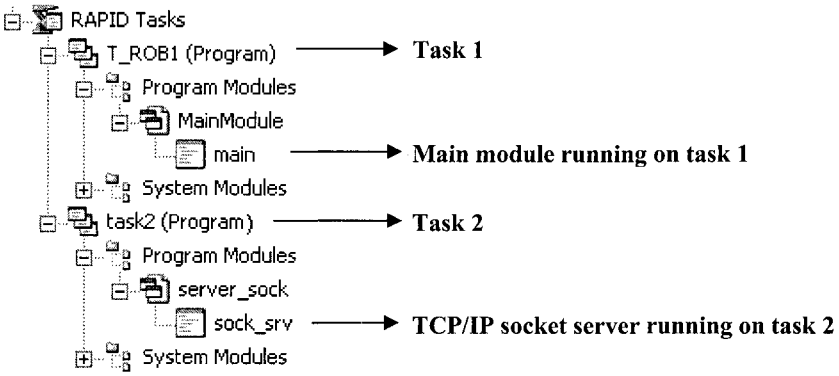


Figure 3.19 View of the tasks available on the system using *RobotStudio Online* (ABB)

The above mentioned applications are different applications in terms of objectives and requirements. Consequently, since the robot control system is a multitasking system, each of them was designed to run in their own task (process) – see Figure 3.19.

A TCP/IP socket server can work like a switch-case-do cycle driven by the received message. The first word of the received message, named "*command*", can be used to drive the cycle and discriminate the option to execute, implementing in this way the services it was designed to offer. Consequently, the TCP/IP server (*sock_srv*, running on task 2) should have a basic structure like the one represented in Figure 3.20.

```
PROC sock_srv()
  SocketCreate server_socket;
  SocketBind server_socket, "172.16.0.89", 2004;
  SocketListen server_socket;
  WHILE TRUE DO
   SocketAccept server_socket, client_socket;
   SocketReceive client_socket \Str := receive_string;
   extract_INFO_from_message (command, parameter{i});
   TEST command
     Case "motor_on"
       motor_on(result);
       SocketSend client_socket, result;
     Case "motor_off"
       motor_off(result);
       SocketSend client_socket, result;
     Case "write_num"
```

```
        write_num(parameter1, parameter2, result);
        SocketSend client_socket, result;
      Case "read_num"
        read_num(parameter1, result);
        SocketSend client_socket, result;
...
    ENDTEST
    SocketClose client_socket;
  ENDWHILE
  ERROR_HANDLER;
ENDPROC
```

Figure 3.20 Basic structure of the TCP/IP socket server running on the robot controller

The server briefly presented in Figure 3.20 implements basically the same functionality listed in Table 3.3. Furthermore, the command strings have a simple structure:

command parameter_1 parameter_2 ... parameter_N

i.e., the command string starts with a word representing the "*command*" (used by the server to discriminate what is the service the user wants to execute), followed by other words corresponding to the "*parameters*" associated with the "*command*". For example:

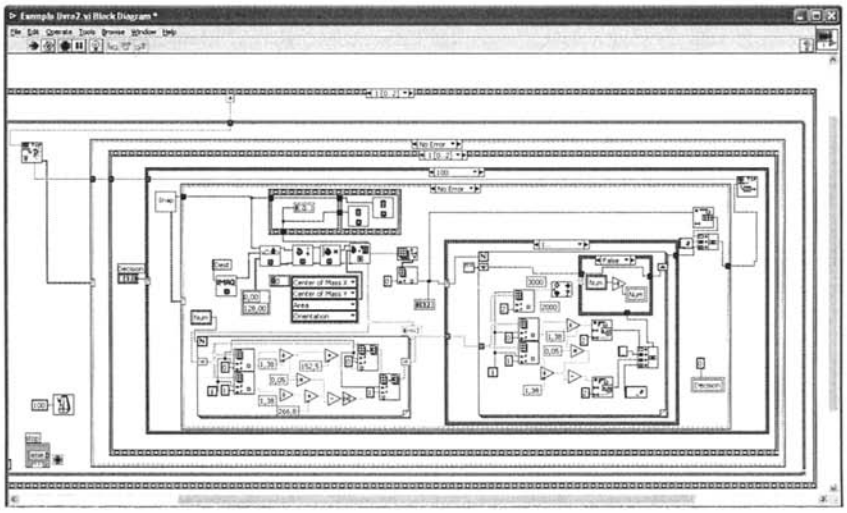
Action	**Command String**
Motor_ON	"*motor_on*"
Motor_OFF	"*motor_off*"
Read_num	"*read_num variable_name*"
Write_num	"*write_num variable_name value*"
Program_start	"*program_start module*"
Program_stop	"*program_stop module*"
...	

where "*variable_name*" is the name of the variable to read, "*value*" is the new value to assign to the variable, and "*module*" is the name of the module to start or stop.

3.4.2 Webcam Software

The application designed to handle the Webcam (Figure 3.17) also works as a TCP/IP server. The reason is simple, the Webcam works here as a sensor used to obtain two types of information: the number of objects and their respective position. Consequently, it is important to be able to address the sensor as an independent entity on the network, and simply command it to return the required information. One simple way to do that is to also adopt a client-server model for

the Webcam software, using TCP/IP sockets to implement it. The software development package used here to add image processing capabilities to the developed software was *LabView* from *National Instruments*. Consequently, the complete application was built on *Labview*, including the TCP/IP socket implementation (Figure 3.21).



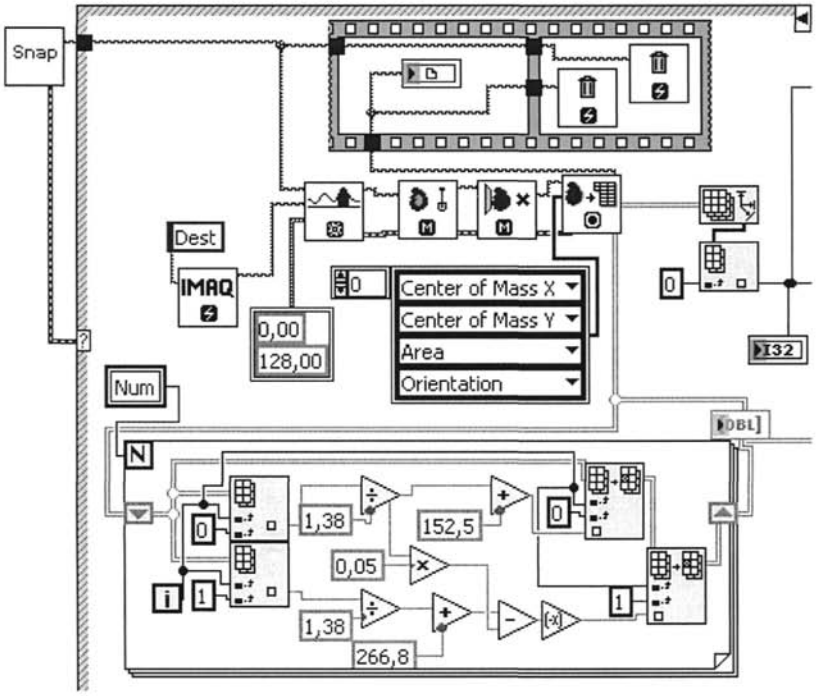


Figure 3.21 *Labview* Vi of the Webcam software (using IMAQ for *LabView*): a – complete VI; b – detail of part of the VI (feature computation)

The Webcam used here is a simple commercial USB Webcam (Figure 3.22) which must be installed on the machine running the above *Labview* mentioned Webcam application.

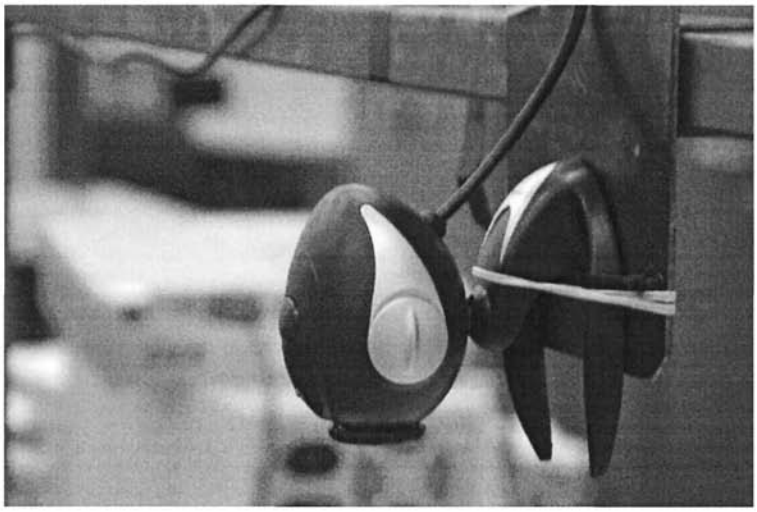


Figure 3.22 Webcam used in this application (*i-C@AM from Liftech Inc.)*

The TCP/IP server handling the Webcam software listens for commands on a specified IP address and port number. When a connection is accepted, the server responds to the following command:

Command - "*camera get objects*"

After receiving the command correctly the server acquires a frame from the Webcam and runs the image processing routine developed for this application. The routine identifies the objects in the captured frame, and for each object computes the center of mass. The TCP/IP client receives the following information:

- Number of objects identified
- Center of mass of each of the identified objects

The answer is sent through the open socket on a string with the following syntax:

number_#x1_y1#x2_y2#...#xN_yN#

where "*number*" is the number of objects identified and *(xi, yi)* is the position of each of the objects. For example, for the scene presented in Figure 3.18:

command from client: "*camera get objects*"
answer from server: *4_#x1_y1#x2_y2#x3_y3#x4_y4#*

3.4.3 Remote Client

The objective of this application is to implement the human-machine interface with the user, providing the resources to enable the user/programmer to command the robot to pick-and-place the existing objects identified by the software associated with the Webcam. Basically, the application can run on any machine with access to the network. For this particular application, a Pocket PC (PPC) running *Windows Mobile 2005* was chosen since the PPC platform is powerful and very interesting for portable HMI applications, namely when a wireless network is available (Figure 3.23).

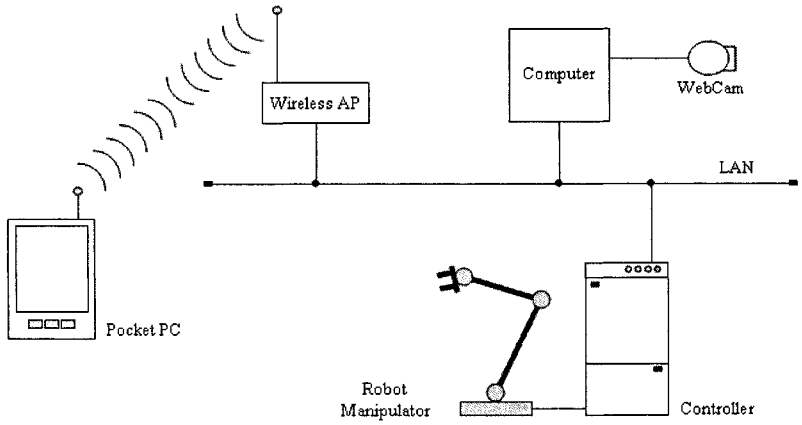


Figure 3.23 Overview of the setup used in this application

In the following material, the code of the client application will be briefly presented, showing in detail a few selected and representative functions. Figure 3.24 shows the screen of the developed PPC application used to connect to the TCP/IP server running on the robot controller and change the robot operating state.

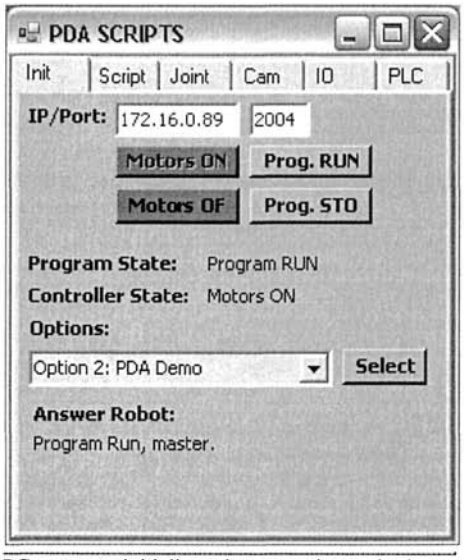


Figure 3.24 PPC screen to initialize robot operation and select program option

This is the code associated with the action "Motors ON" (Figure 3.24):

```
server_name = ip.Text
server_port = port.Text
sock = ConnectSocket(server_name, server_port)
If sock Is Nothing Then
   ans_robot.Text() = "Error connecting to robot, master."
Else
   Dim bytesSent As [Byte]() = Nothing
   bytesSent = ascii.GetBytes("motor_on")
   sock.Send(bytesSent, bytesSent.Length, 0)
   bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0)
   ans_robot.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
   moff.Enabled = True
   mon.Enabled = False
   prun.Enabled = True
   pstop.Enabled = True
   sel.Enabled = True
   sock.Close()
   If Encoding.ASCII.GetString(bytesReceived, 0, bytes) = "0" Then
      ans_robot.Text() = "Motor on, master."
      cstate.Text() = "Motors ON"
   Else
      ans_robot.Text() = "Error executing, master."
   End If
End If
```

The code presented above simply opens the socket, sends the commanding string, and processes the answer. This code is associated with the software button "*Motor ON*" in Figure 3.24.

To give another example, the code associated with the action "*Program RUN*" (Figure 3.24) is presented below:

```
Server_name = ip.Text
server_port = port.Text
sock = ConnectSocket(server_name, server_port)
If s Is Nothing Then
   ans_robot.Text() = "Error connecting, master."
Else
   Dim bytesSent As [Byte]() = Nothing
   bytesSent = ascii.GetBytes("program_start_main")
   sock.Send(bytesSent, bytesSent.Length, 0)
   bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0)
   ans_robot.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
   sock.Close()
   If Encoding.ASCII.GetString(bytesReceived, 0, bytes) = "0" Then
      ans_robot.Text() = "Program Run, master."
      pstate.Text() = "Program RUN"
   Else
      ans_robot.Text() = "Error executing, master."
   End If
End If
```

The interface with the Webcam is done through the screen window represented in Figure 3.25. Using this window, the user can command the camera to return the information about the objects in the scene. All the returned positions are listed in the list-box present in the interface (Figure 3.25) for the user to select the one he wants to use for the pick-and-place operation.

The code below details the implementation of the action "*Get Webcam Picture*" (Figure 3.25):

```
Dim msg_received As String
Dim indx As Integer
Dim num_obj As Integer
Dim index As Integer
sock = ConnectSocket(ip2.Text.ToString, port2.Text.ToString)
If sock Is Nothing Then
   ans_robot_3.Text() = "Error connecting to CCD, master."
Else
   Dim bytesSent As [Byte]() = Nothing
   bytesSent = ascii.GetBytes("camera get objects")
```

```
    If s.Available <> 0 Then
      bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0)
      MsgBox("ok, buffer cleared.")
    End If
    sock.Send(bytesSent, bytesSent.Length, 0)
    bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0)
    list_cam.Items.Clear()
    msg_received = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
    If msg_received <> "0_#no objects" Then
      indx = msg_received.IndexOf("#")
      num_obj = msg_received.Substring(0, indx - 1)
      n_obj.Text() = num_obj
      msg_received = msg_received.Substring(indx + 1)
      For index = 1 To (num_obj - 1) Step 1
        indx = msg_received.IndexOf("#")
        object_cam(index) = msg_received.Substring(0, indx - 1)
        list_cam.Items.Item(index - 1) = object_cam(index)
        msg_received = msg_received.Substring(indx + 1)
      Next
      index = num_obj
      indx = msg_received.IndexOf("#")
      object_cam(index) = msg_received.Substring(0, indx - 1)
      list_cam.Items.Item(index - 1) = object_cam(index)
    Else
      ans_robot_3.Text() = "no objects"
    End If
    sock.Close()
  End If
```

In the code above, the information about the number and position of the identified objects is extracted from the returned string and listed in the list-box and other output textboxes. The user can then select one of the obtained positions and command the robot to pick that object and place it on the output container box. The code below is the implementation of the "*Pick*" action (Figure 3.25):

```
  sock = ConnectSocket(ip2.Text.ToString, port2.Text.ToString)
  Pick.Enabled = False
  If sock Is Nothing Then
    ans_robot_3.Text() = "Error connecting, master."
  Else
    Dim bytesSent As [Byte]() = Nothing
    bytesSent = ascii.GetBytes("command_str 5000_" +
                object_cam(list_cam.SelectedIndex + 1))
    sock.Send(bytesSent, bytesSent.Length, 0)
    bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0)
    ans_robot.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
    sock.Close()
```

```
    If Encoding.ASCII.GetString(bytesReceived, 0, bytes) = "0" Then
      ans_robot_3.Text() = "Pick command, master."
      list_cam.Items.Item(list_cam.SelectedIndex) = "no object"
    Else
      ans_robot_3.Text() = "Error executing, master."
    End If
  End If
```

The "*Pick*" action is associated with a robot subroutine driven by the variable "*command_str*". The action is identified with the number 5000, and requires the user to specify also the parameters X and Y, referring to the position of the object. Consequently, the command from the client application to successfully trigger the "Pick" action is,

```
bytesSent = ascii.GetBytes("command_str 5000_" +
                           object_cam(list_cam.SelectedIndex + 1))
```

which translates to,

command_str 5000 X Y

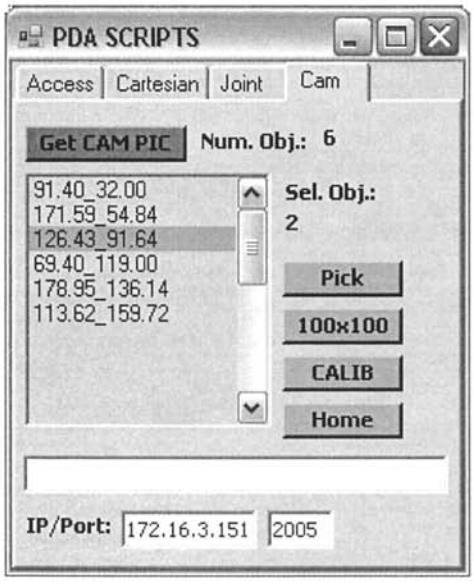


Figure 3.25 PPC screen designed to interface the Webcam

The robot subroutine handles these commands in the way presented below:

```
IF index = receive_len+1 THEN
  command_str:=receive_string1;
ENDIF
IF (index > 1) and (index < receive_len) THEN
  command_str:=StrPart(receive_string1,1,index-1);
  str_aux1:=StrPart(receive_string1,index+1,receive_len-index);
  receive_len:=StrLen(str_aux1);
  index:=StrMatch(str_aux1,1,"_");
  IF index = (receive_len + 1) THEN
    parameter1a:=str_aux1;
  ENDIF
  IF (index > 1) and (index < receive_len) THEN
    parameter1a:=StrPart(str_aux1,1,index-1);
    str_aux2:=StrPart(str_aux1,index+1,receive_len-index);
    receive_len:=StrLen(str_aux2);
    index:=StrMatch(str_aux2,1,"_");
    IF index = (receive_len + 1) THEN
      parameter2a:=str_aux2;
    ENDIF
  ENDIF
  IF (index > 1) and (index < receive_len) THEN
    parameter2a:=StrPart(str_aux2,1,index-1);
    str_aux3:=StrPart(str_aux2,index+1,receive_len-index);
    receive_len:=StrLen(str_aux3);
    index:=StrMatch(str_aux3,1,"_");
    IF index = (receive_len + 1) THEN
      parameter3a:=str_aux3;
    ENDIF
    IF (index > 1) and (index < receive_len) THEN
      parameter3a:=StrPart(str_aux3,1,index-1);
    ENDIF
  ENDIF
ENDIF
TEST command_str
 case "190": movecontact;
 case "200": open_g;
 case "201": close_g;
 case "301": move_P1;
 case "401": go_home;
 case "501": movej1p;
 case "502": movej1m;
 case "503": movej2p;
 case "504": movej2m;
 case "505": movej3p;
 case "506": movej3m;
```

```
  case "507": movej4p;
  case "508": movej4m;
  case "509": movej5p;
  case "510": movej5m;
  case "511": movej6p;
  case "512": movej6m;
  case "520": jammount1;
  case "530": cammount1;
  case "540": pick_pen;
  case "550": release_pen;
  case "1000": save_pos;
  case "2000": move_table;
  case "3000": exe_script;
  case "5000": cam_pick;
  case "5001": cam_go;
 ENDTEST
```

Basically, the routine extracts the information from the command string sent through the socket connection, and feeds the controlling variables with the commanded values. The TEST cycle (similar to a switch-case-do cycle) discriminates the function to call, which executes the functionality commanded by the user.

This example shows in some detail the procedure to explore TCP/IP socket servers for industrial manufacturing systems. It also shows that there are several platforms available to simplify the HMI and the setup, making the overall application easier to use.

3.4.4 Using UDP Datagrams

Using UDP datagrams (socket datagrams) is not fundamentally different than using TCP sockets (stream datagrams). Consequently, a simple implementation is mentioned here with the objective of pointing out the practical. The selected implementation uses a MOTOMAN robot (model HP6) equipped with the new NX100 robot controller. This controller offers remote services available from a UDP socket server, which are similar in functionality to the ones listed in Table 3.3. Several client applications were developed by the author to access those services, including the secondary services built based on those available from the UDP server, using *the Microsoft Visual Studio .NET 2005* programming suite. In the following, a simple application developed to run on Pocket PC (running *Windows Mobile 2005*) will be briefly introduced.

When using UDP datagrams, which are unreliable connections, the user should not use blocking calls, i.e., connections that block the application while waiting on the socket for the answer to the call. Consequently, after opening a socket and sending a UDP datagram, the user program shouldn't wait forever for an answer on the

socket or thread. Instead, it should close the socket based on a timeout event. The following application (Figure 3.26) runs on PPC and makes a few UDP datagram calls to the UDP socket server running on the robot controller.

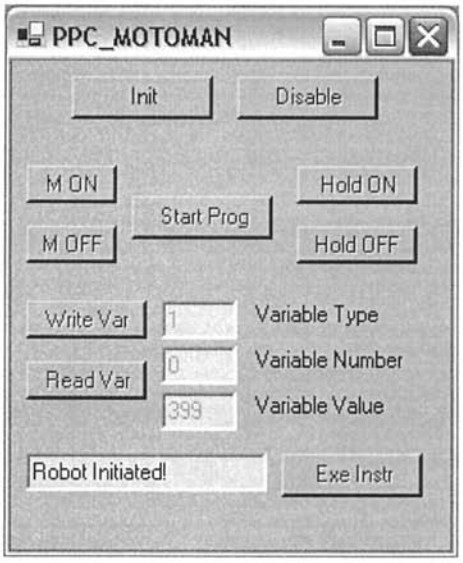


Figure 3.26 PPC application designed for a *Motoman* robot to explore UDP services from its NX100 controller

The program running on the robot controller, to implement operational (or secondary) services, is a switch-case-do type cycle driven by a numeric variable (type 1, index 0 – in the motoman notation). The simple server for this application moves the robot to five fixed positions, depending on the value of the above mentioned variable:

```
WHILE never_end
  WAIT B00 <> 0;
  TEST B00
    Case 399
      MOVE P1, VEL, 0, T0;
    Case 499
      MOVE P2, VEL, 0, T0;
    Case 599
      MOVE P3, VEL, 0, T0;
    Case 699
      MOVE P4, VEL, 0, T0;
    Case 799
      MOVE P5, VEL, 0, T0;
  ENDTEST
```

```
  B00 = 0;
RETURN
```

Writing, for example, the value 399 in the variable B00 makes the robot move to position P1. The code associated with requesting that action remotely is:

```
Dim remoteIP As New IPEndPoint(IPAddress.Parse("172.16.0.93"), 10006)
Dim Socket_send As New Socket(remoteIP.AddressFamily, SocketType.Dgram,
ProtocolType.Udp)
Dim Socket_receive As New UdpClient(10006)
Dim ENQ() As Byte = {&H6, &H0, &H1, &H0, &H5}
Dim EOT() As Byte = {&H6, &H0, &H1, &H0, &H4}
Dim ACK0() As Byte = {&H6, &H0, &H2, &H0, &H10, &H30}
Dim ACK1() As Byte = {&H6, &H0, &H2, &H0, &H10, &H31}
Socket_send.Connect(remoteIP)
Socket_receive.Connect(remoteIP)
Dim str_temp As String
Socket_send.Send(ENQ)
Dim receiveBytes As [Byte]() = Socket_receive.Receive(remoteIP)
recb = receiveBytes.Length()
For i As Integer = 0 To recb - 1
  str_temp = str_temp + Hex(receiveBytes(i))
Next i
If str_temp <> "60201030" Then
  MessageBox.Show("Erro na resposta ao ENQ: " + str_temp)
  Socket_send.Close()
  Socket_receive.Close()
  Return
End If

Dim str_temp As String
Socket_send.Send(Comando)
Dim receiveBytes As [Byte]() = Socket_receive.Receive(remoteIP)
recb = receiveBytes.Length()
For i As Integer = 0 To recb - 1
  str_temp = str_temp + Hex(receiveBytes(i))
Next i
If str_temp <> "60201031" Then
  MessageBox.Show("Erro na resposta ao comando: " + str_temp)
  Socket_send.Close()
  Socket_receive.Close()
  Return
End If
...
Send End Of Transmission
Send ACK0
Send ACK1
```

...
Socket_send.Close()
Socket_receive.Close()
This code is rather complex, since all the details about the protocol, including the negotiation phases, are explicitly programmed in the function. Basically, to send a command the protocol adopted by *Motoman* requires a command start, followed by the command itself, and then an end-of-command sequence.

The reader should remember that the sockets named "*socket_receive*" have a pre-defined timeout that prevents the application from blocking. When a timeout occurs, the routine returns immediately.

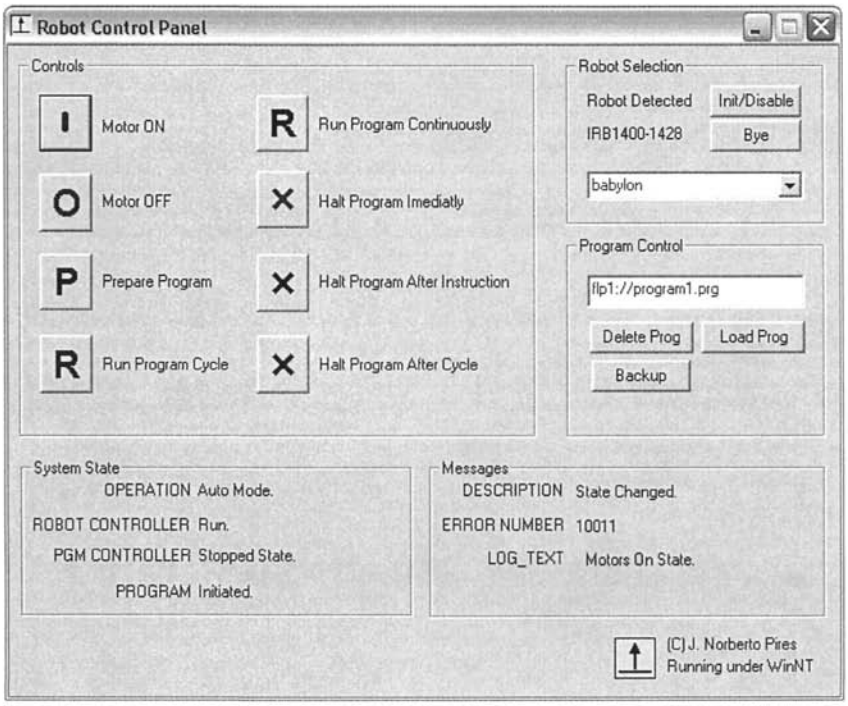


Figure 3.27 Control panel application events ("*messages*") received from the robot controller

3.5 Simple Example: Control Panel

The "*Control Panel*" is rather different from the previous examples. First, it uses *remote procedure calls* (RPCs) to access the services available from the remote server, which is a standard way to offer services and to support client-server programming environments. Other than that, the application works also as an RPC

server, because it is capable of receiving events from the robot controller. The events are RPC calls made by the controller to the machines that made subscriptions to receive those events.
The application was built using PCROBNET2003/5 [5-7], an *ActiveX* software component that offers the methods, properties, and data structures necessary to explore the RPC services from the robot controller (ABB S4 robot controllers).The code for some selected actions is briefly explored below. For example, the code (developed in C++ using methods from the above mentioned *ActiveX* component) for the actions "*MOTOR ON*", "*MOTOR OFF*", "*PROGRAM RUN*", and "*PROGRAM STOP*" is presented below:

```
void CCtrpanelDlg::Onmotoron()
{
 nresult = m_pon.MotorON();          <-------------------  Call method
 if (nresult == -8999) no_comms = TRUE;
}

void CCtrpanelDlg::Onmotoroff()
{
 nresult = m_pon.MotorOFF();         <-------------------  Call method
 if (nresult == -8999) no_comms = TRUE;
}

void CCtrpanelDlg::Onrunprogramcon()
{
 long cycles = -1;
 long mode = 1;
 nresult = m_pon.ProgStart("main",&cycles, &mode);  <----  Call method
 if (nresult == -8999) no_comms = TRUE;
}

void CCtrpanelDlg::Onhaltprogramim()
{
 short mode = 3;
 nresult = m_pon.ProgStop(&mode);    <-------------------  Call method
 if (nresult == -8999) no_comms = TRUE;
}
```

To receive events, a specially developed RPC server must be running on the *client* computer to receive those RPC calls. That server broadcasts the received events as registered operating system user messages (Figure 3.13). Consequently, to be able to receive those events, each application just needs to watch its message queue and filter the relevant messages. The code below was designed to operate on the message queue to identify events and present the information to the user (see "*messages*" in Figure 3.27).

```
void CCtrpanelDlg::OnSponMsgPcroB.C.trl1(long FAR* msg_number, long
FAR* msg_lParam, long FAR* msg_wParam)
{
 BSTR msg;
 m_pon.ReadMsg(&msg, msg_lParam, msg_wParam);
 CString Msg(msg);
 m_logtext.SetWindowText(Msg);
 SysFreeString(msg);
 switch (*msg_lParam)
 {
   case 1: m_description.SetWindowText("State Changed."); break;
   case 2: m_description.SetWindowText("Warning."); break;
   case 3: m_description.SetWindowText("Error."); break;
   default: m_description.SetWindowText("Invalid log_type."); break;
 }
 Msg.Format("%d",*msg_wParam);
 m_error.SetWindowText(Msg);
 CCtrpanelDlg::info();
}
```

Using software components (*ActiveX*, *JAVA*, etc.) is a way to hide from the user the tricky details about how to make RPC calls (for example, compare this code with the one presented for the UDP datagram example), allowing her to focus immediately on the application.

3.6 Simple Example: S4Misc – Data Access on a Remote Controller

The "*S4Misc*" application (Figure 3.28) also uses RPC to access the robot services. Like the previous example, it was designed to be used with the ABB S4 robot controllers (running option RAP [8]).

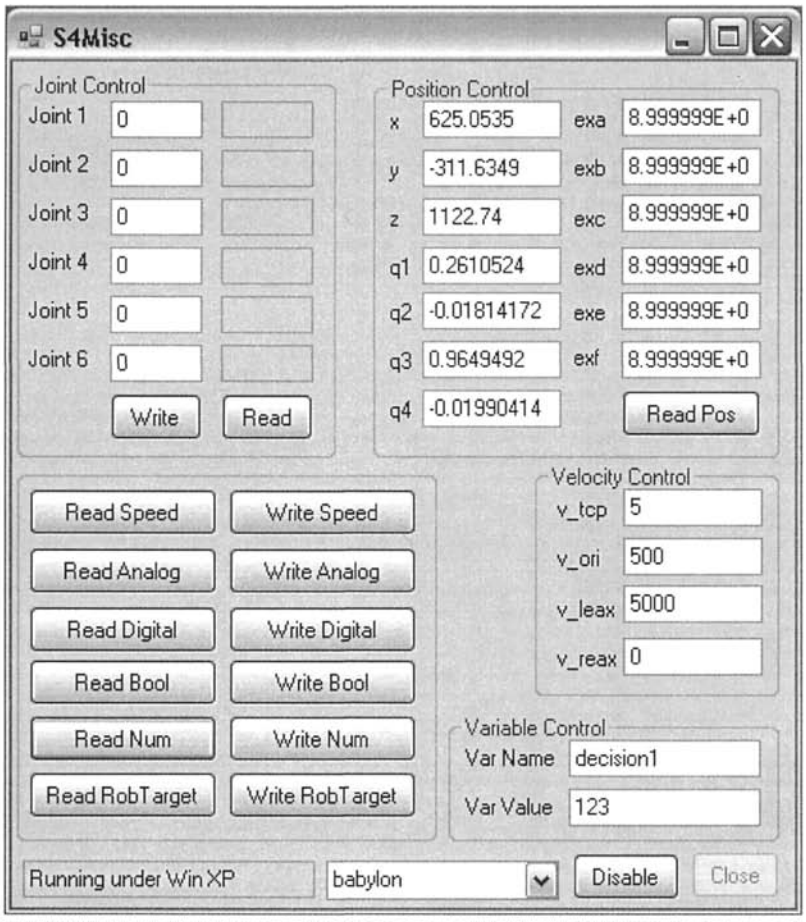


Figure 3.28 *S4Misc* application designed to access program and system variables from a remote computer

This application enables the user to access program and system variables from a remote computer online, i.e., even when the robot is in automatic mode and the loaded program is executing. The user can utilize this software for debugging purposes, checking and changing (when needed) the actual value of any variable. In the following, the code for the actions *READ/WRITE* a numeric variable, *WRITE* a speed variable, and *READ* the actual robot position is showed (*C# .Net 2005* was used here):

```
private void OnReaNum()
{
   String msg;
   msg = txt_VarName.Text;
   if (msg.Length > 0)
```

```
    {
      nresult = PcRob.ReadNum(msg, ref val);  <------ Call method
      if (nresult < 0)
      {
        MessageBox.Show("Error Reading Num!");
      }
      else
      {
        msg = Convert.ToString(val);
        txt_VarValue.Text = msg;
      }
    }
    else MessageBox.Show("Error: You must specify variable name!");
}

private void OnWriteNum()
{
    String msg;
    String msg1;
    msg = txt_VarName.Text;
    msg1 = txt_VarValue.Text;
    if (msg.Length > 0 || msg1.Length > 0)
    {
      val = Convert.ToSingle(msg1);
      nresult = PcRob.WriteNum(msg, ref val);  <------ Call method
      if (nresult < 0) MessageBox.Show("Error Wrinting Num!");
    }
    else MessageBox.Show("Error: You must specify variable name and value!");


private void OnWriteSpeed()
{
    String msg;
    msg = txt_VarName.Text;
    if (msg.Length > 0)
    {
      RobVelocity.vtcp = Convert.ToSingle(txt_VTcp.Text);
      RobVelocity.vori = Convert.ToSingle(txt_VOri.Text);
      RobVelocity.vleax = Convert.ToSingle(txt_VLeax.Text);
      RobVelocity.vreax = Convert.ToSingle(txt_VReax.Text);
      PcRob.vtcp = RobVelocity.vtcp;
      PcRob.vori = RobVelocity.vori;
      PcRob.vleax = RobVelocity.vleax;
      PcRob.vreax = RobVelocity.vreax;
      nresult = PcRob.WriteSpeedDataVB(msg);  <------ Call method
      if (nresult<0) MessageBox.Show("Error: You must specify variable name");
    }
```

```
    else MessageBox.Show("Error: You must specify variable name");
}

private void OnReadCurrRoboTarget()
{
    nresult =PcRob.ReadCurrRobTVB(); ◄—— Call method
    if (nresult < 0)
    {
        MessageBox.Show("Error Reading Current RobT");
    } else
    {
        RobT_Read.x = PcRob.x;
        RobT_Read.y = PcRob.y;
        RobT_Read.z = PcRob.z;
        RobT_Read.q1 = PcRob.q1;
        RobT_Read.q2 = PcRob.q2;
        RobT_Read.q3 = PcRob.q3;
        RobT_Read.q4 = PcRob.q4;
        RobT_Read.exa = PcRob.exa;
        RobT_Read.exb = PcRob.exb;
        RobT_Read.exc = PcRob.exc;
        RobT_Read.exd = PcRob.exd;
        RobT_Read.exe = PcRob.exe;
        RobT_Read.exf = PcRob.exf;
        txt_x.Text = RobT_Read.x.ToString();
        txt_y.Text = RobT_Read.y.ToString();
        txt_z.Text = RobT_Read.z.ToString();
        txt_q1.Text = RobT_Read.q1.ToString();
        txt_q2.Text = RobT_Read.q2.ToString();
        txt_q3.Text = RobT_Read.q3.ToString();
        txt_q4.Text = RobT_Read.q4.ToString();
        txt_exa.Text = RobT_Read.exa.ToString();
        txt_exb.Text = RobT_Read.exb.ToString();
        txt_exc.Text = RobT_Read.exc.ToString();
        txt_exd.Text = RobT_Read.exd.ToString();
        txt_exe.Text = RobT_Read.exe.ToString();
        txt_exf.Text = RobT_Read.exf.ToString();
    }
}
```

This application demonstrates the usefulness of having remote services that can communicate with the running applications. With it, users can influence the behavior of running applications for controlling, monitoring, or debugging purposes. It also demonstrates the usefulness of software components for the process of developing distributed applications that necessarily use several types of radically different equipment. With these components, users and programmers can

focus on the applications under development without worrying about the technical details of remote procedure calling, network communications, and so on.

3.7 Industrial Example: Semi-autonomous Labeling System

In this section, an industrial example that explores the previous material is presented and discussed. This example corresponds to an actual implementation resulting from a cooperation effort between the author and a Portuguese company. The system presented here was designed to operate almost without operator intervention, showing that concepts like flexibility and agility are fundamental to manufacturing plants and require much more from the systems used on the shop floor. Flexible manufacturing systems take advantage of being composed of programmable equipment to implement most of its characteristics, which are supported by reconfigurable mechanical parts. Industrial robots are, consequently, good examples of flexible manufacturing systems.

The robotic industrial system presented here was designed to execute parameterized labeling tasks on paper rolls. The system is commanded directly from the manufacturing tracking and control software. This software is based on dynamic databases that register the situation of each item produced in the factory, a simple way to track them see what is happening on the shop floor. Since all information about each item is available in the manufacturing tracking software, it is logical to use it to command some of the shop floor manufacturing systems, namely the ones that require only simple parameterization to work properly. This procedure would take advantage of the available information and infrastructure, avoiding unnecessary operator interfaces to command the system. Also, potential gains in terms of flexibility and productivity are evident.

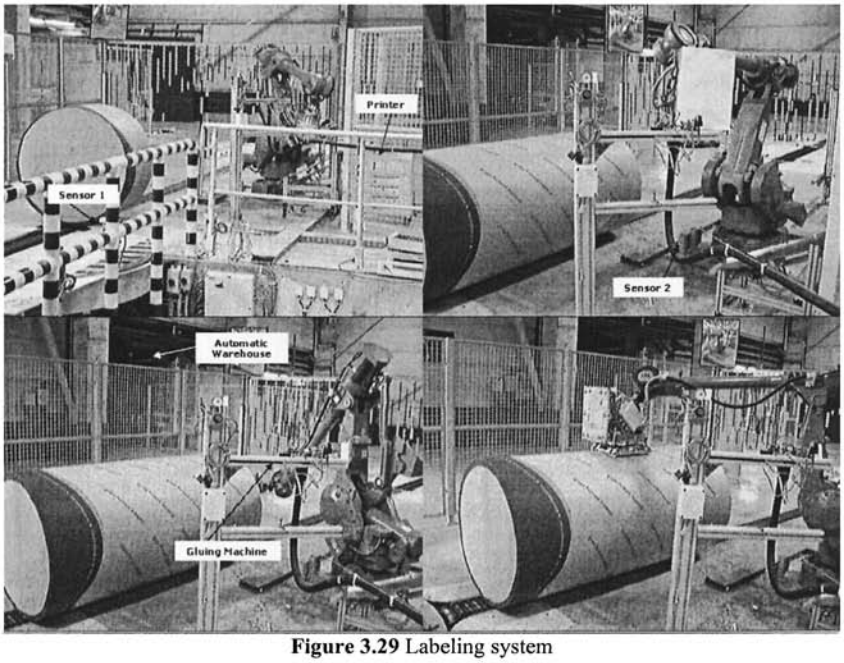


Figure 3.29 Labeling system

3.7.1 Robotic Labeling System

The industrial system introduced here is a labeling system (Figure 3.29) composed of:

- One robot manipulator ABB IRB4400, with the S4C+ controller [10]
- One electro-pneumatic gripper, properly equipped to grab one or two A4-size paper sheets
- One office laser printer, with several trays of paper
- One gluing machine with spray injectors controlled from the robot controller IO system
- One industrial PLC (Siemens S7-300) that controls the rolls conveyer belt, providing information to the robot controller about its state

In general, the labeling robotic system works as follows: When a roll is released from the previous system (wrapping machine), one or two labels are printed on the laser printer. At the same time, the robot receives the order to pick those labels from the ramp placed at the end of the printer, and immediately prepositions near the printer. The picking operation happens when the required number of sheets are available at the ramp (two optical sensors detect the presence of paper). After that, the robot waits for the roll to enter the working zone, i.e., waits for the corresponding optical sensor, named sensor 1 in Figure 3.29, to detect the roll. When the roll is detected, the robot moves to the gluing machine to add glue on the

side of each label. When the operation is finished, the roll should be already stopped, waiting for the robot to insert the labels on the top and on the right side of the roll. The robot performs that operation when the roll is detected by sensor 2 (Figure 3.29) and when the PLC confirms that the conveyor has stopped. When the operation is finished, the robot signals it using a flag, accessible remotely, and moves to a neutral position to wait for a new command.

3.7.2 System Software

Designing software for the system, which needed to be commanded from the network, was an interesting challenge. The industrial robot is the central element of the manufacturing cell, and is connected to the factory network, which makes it easily accessible from the UNIX station running the manufacturing tracking software.

To exchange information between computer systems, in a safe and guaranteed way, a client-server approach using TCP/IP sockets may be used. That is a simple and straightforward thing to do, with the UNIX computer acting as the client. A TCP/IP server should then be available to receive client calls, and a properly designed messaging protocol must be used. The decision here was to make the TCP/IP server the only interface to the robotic manufacturing cell, so that any command or request of information is done by connecting to the server and sending the appropriate messages. Since there is a network on the shop floor, the TCP/IP server can be installed in any shop floor computer, making it really easy to install the interface and have it running. In the factory under consideration, the majority of the shop floor computers are running the *Windows NT4* and *Windows 2000* operating systems. Consequently, we decided to use BSD compatible TCP/IP sockets, which are also compatible with the Microsoft TCP/IP implementation (*winsock2*).

The next challenge was how to manage the communication with the robot controller, since it is well known that actual robot controllers are closed industrial systems not allowing installation of any user software apart from robot programs. ABB robot controllers [10] have internal *Remote Procedure Call* (RPC) [2,8] servers that can be used to exchange variables, files, etc. Those servers are *SUN RPC 4.0* [2] compatible, and can be used to our purposes if the TCP/IP server interface can issue RPC calls to the robot controller. Consequently, a library of functions implementing calls for all the services on the ABB robot controller was built [5,7], along with a port of the *SUN RPC 4.0* to operating systems based on the *Win32* API. This environment enables a complete access to the robot controller RPC services making it possible to command the robot from the network. The robot controller software must then be built in a way to expose all system capabilities to the remote client. This means building it like a *SWITCH-CASE-DO* server, with the switching variable controlled by the remote client.

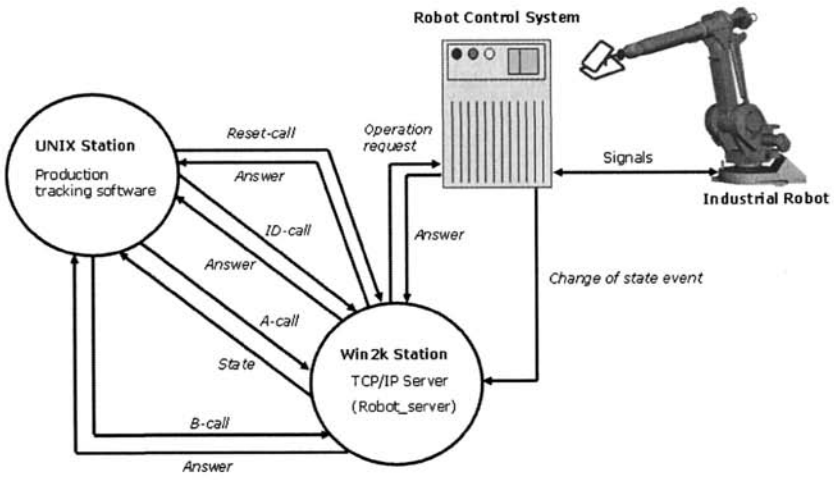


Figure 3.30 Software interface to the industrial robotic system

The basic idea, depicted in Figure 3.30, is simple. The interface to the industrial robotic cell is a TCP/IP server running on a specified IP and port number. The following procedure is used in a way to guarantee safety and avoid data loss:

- The server should respond to *ID-calls* with a pre-determined string, which is used to identify the TCP/IP server with name, version, and date. The string is actually "*robot_server@v21m11y03*". The *ID-call* is the first call issued by the client after establishing a new connection. A wrong answer to the *ID-call* should tell the client to send a *reset-call* and close the connection
- The client makes frequent *A-calls*, in periods of two seconds, to find out if the server is alive and healthy, and to get its actual state (*busy* or *ready*)
- The client uses *B-calls* to send execution commands, properly parameterized, to the robotic labeling system. When a *B-call* is received and accepted by the server, the system enters the *busy* state and any subsequent *A-call* will return that the system is *busy*
- When the robotic labeling system completes a task, i.e., when it inserts the requested number of labels on the roll in use, the system enters the *ready* state and any subsequent *A-call* will return that state

The TCP/IP server is the only operational interface to the robotic system. Basically, it is a simple single channel TCP/IP server, completely coded in C++, which waits for connections on a pre-determined port, accepting only the ones coming from only a few IPs (the ones where the manufacturing tracking software may be running). Connection is established only if the calling machine makes an *ID-call*, properly parameterized, including a password. The server is a state machine that implements answers to the four different messages that can be sent by

the connected client (Figure 3.30). The connection between the TCP/IP server and the industrial robot is handled using RPC sockets, compatible with the SUN RPC 4.0 definition.

In the following section, the developed software will be further explained, starting with the software designed to run on the robot controller.

3.7.3 Robot Controller Software

Considering that the system was designed to be commanded remotely using the factory computer network, it was decided to have the robot controller software working as a server, exposing to the remote client all of its operational functionalities. This capability is very interesting also for other applications, and because of that it will be discussed in a general way.

When building an industrial robotic cell, it is certainly possible, and very useful, to identify all the system capabilities and requirements, i.e., the system engineer should state clearly all the functions it is supposed to perform and write the code necessary to implement them. If that code is developed as general as possible, and used to build a server that can be explored remotely with properly parameterized calls, then the complete system functionality can be requested remotely from the network.

Technically, to implement the remote calls, it was decided to *use remote procedure calls* (RPC) compatible with the *SUN RPC 4.0* suite, an open standard in the public domain. The ABB S4 robot control system implements a collection of RPC services that enable users to access programs, system data, and robot configuration, as well as to share files, etc. These services are part of the robot controller's operating system. Using those services from the TCP/IP server designed to interface the system [2, 5-8], it is certainly possible to set up an RPC-driven server like:

```
switch service_decision_variable
  case 1: call function_1; break;
  case 2: call function_2; break;
  case 3: call function_3; break;
  ...
  case n: call function_n; break;
  default: call invalid_function; break;
end_switch
```

where the *service_decision_variable* is a numerical variable whose value can be changed remotely, making an RPC call to the *change_numerical_value* service. In this way, the robot's operation is completely controlled from the remote client.

Since the robotic system is to be operated without human intervention, a few services were added to allow maintenance and error recovery operations. Sometimes, due to errors in the manufacturing tracking database (usually introduced by human intervention), invalid or badly parameterized commands are sent to the robot. In those situations, depending on the dimensions of the roll in use, the robot may crash with the surface of the roll, because it uses the commanded dimensions to approach the surface of the roll in a more efficient way. Also, failure in the conveyor sensors or actuators may cause problems with roll placement. In any case, an operator is required to solve the problems and put the system in production again. The program shown in Figure 3.31 is used to put the robot in a known position and resume automatic operation.

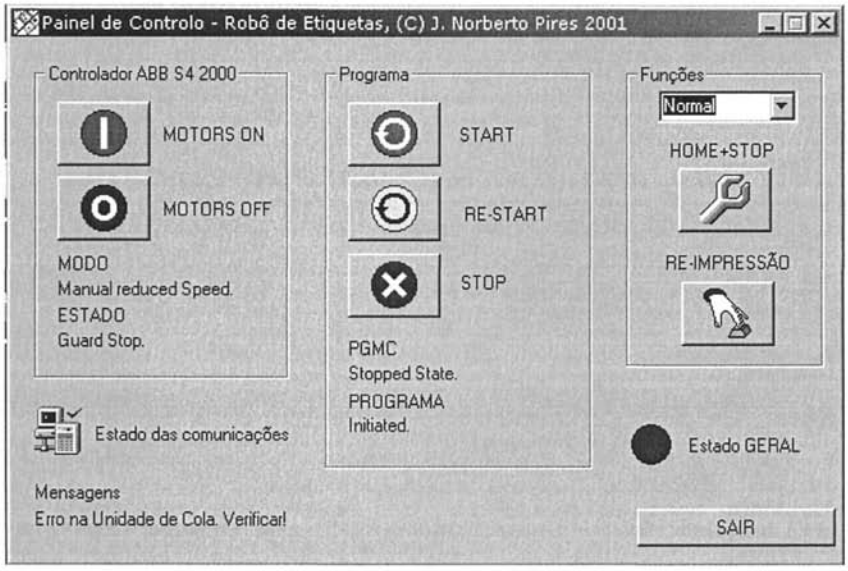


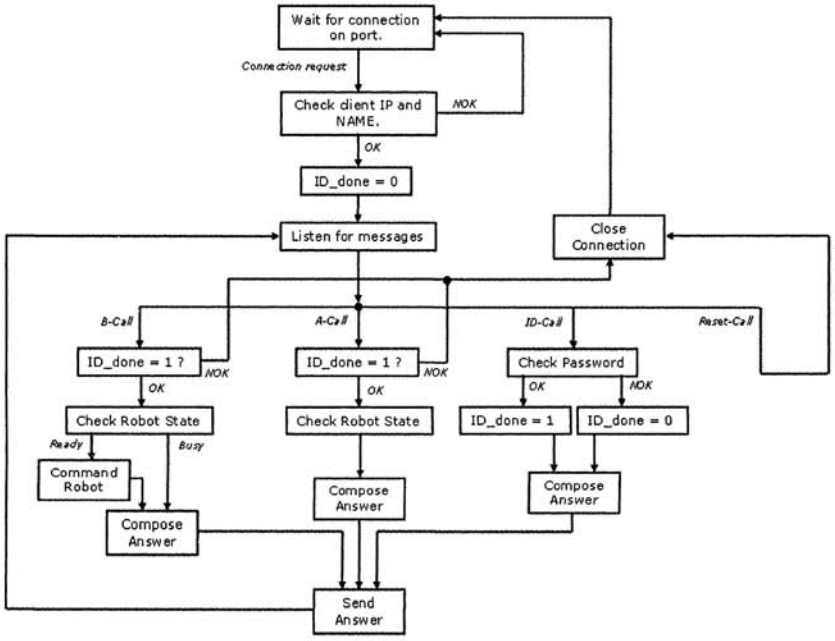
Figure 3.31 Operator interface used to solve error situations

Basically, the operator interface makes RPC calls to the above mentioned services designed to solve erroneous situations. Those services enable the operator to resume local program execution from an actual point or from the beginning, move the robot to well-known positions, enter maintenance routines, and so on. The program usually runs on a laptop that maintenance personnel carry to the setup when a problem arises, plugging it to the network.

3.7.4 TCP/IP Server

As already explained, this TCP/IP server (Figure 3.32) was developed as the only interface to the robotic labeling system. It is a simple TCP/IP server that accepts connections coming from the machine that runs the manufacturing tracking

software (client). After connecting, it implements a state-machine that listens for messages coming from the client, acting accordingly. The TCP/IP server monitors the connection to the robot and the robot state, so that proper answers are given to every *A-call* received from the client. Also, the server does not accept any command in the periods where the robot state is *busy*, forcing the client to wait until the previous commanded operation finishes.



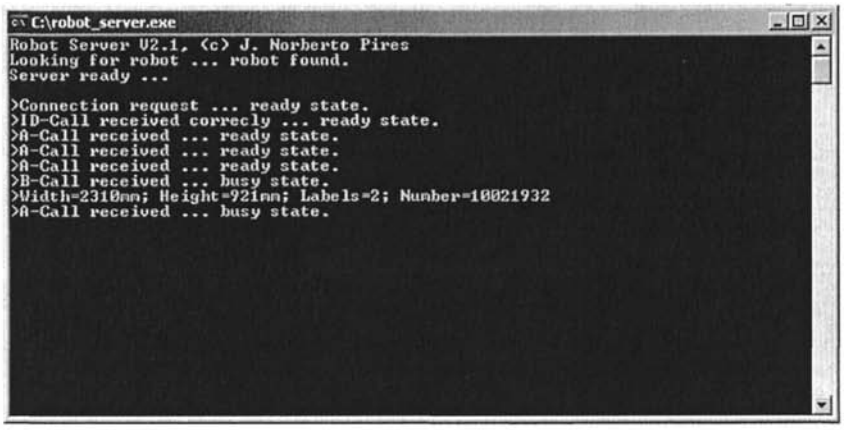


Figure 3.32 TCP/IP server operation

3.7.5 Discussion

The example presented in this section explores the use of software interfaces for remote command of shop floor industrial robotic cells. This involves network interfaces based on the TCP/IP protocol and remote procedure calls, enabling direct command of shop floor manufacturing setups from anywhere in the factory. This idea is particularly useful with systems that require minor parameterization to perform a predefined task. This can easily be done from the manufacturing tracking software, used to follow and manage production, where the require information is available.

In many industries, like the one presented in this example, production is closely tracked in any part of the manufacturing cycle. The manufacturing cycle can be interpreted as a collection of operations and processes that transform the raw materials into finished products. This journey of products between the raw materials warehouse and the finished products warehouse, is composed of several manufacturing systems that perform the necessary operations on the product under processing, and intermediate buffers used to temporarily store semi-finished products in several stages of their production cycle. These buffers are fundamental for a well balanced production planning, achieving high rates of efficiency and agility. In many cases, the manufacturing systems require only minor parameterization to execute their tasks. If that parameterization can be commanded remotely from where it is available (using manufacturing tracking software), then the system becomes almost autonomous in the sense that operator intervention is minimal and related only with maintenance and error situations. A system like this will improve manufacturing efficiency and agility, since the operation becomes less dependent on operators. Also, because the system was built to be explored remotely, which requires a collection of general software routines designed to implement all of the system functionalities, it is easier to change production by changing parameterization, a software task, which also contributes to agility.

This robotic manufacturing system uses a simple TCP/IP server as the commanding interface. The server sends remote procedure calls to the robot control system, which is the system main computer. The robot controller interfaces with the system PLC that controls the conveyor, and manages the information coming from manual controls and sensors. Consequently, any client connected to the TCP/IP server is able to command the system and get production information. This feature adds flexibility and agility to the manufacturing setup. This setup was installed in a Portuguese paper factory and is being used without problems for almost three years, which demonstrates its robustness and simplicity.

Finally it is worthwhile to stress that:

- The system interface was implemented in C++ using available programming tools: *Visual C++ 6.0* first, and *Visual .NET 2003* when an update was necessary [11]

- The system was implemented using standard operating systems, namely, *UNIX* from *Digital* to run the manufacturing tracking software, and *Windows 2000* to run the robotic cell TCP/IP interface
- The *Microsoft* TCP/IP socket implementation was used to program the TCP/IP server, since it is BSD-compatible
- The system uses RPC's compatible with *SUN RPC 4.0*, an open standard not compatible with the *Microsoft* implementation, which required a complete port to *Windows 2000* (the operating system used on the shop floor of the partner factory). That effort was completely done by the author

Consequently, no special tools were used to build the presented solution, which proves that these techniques are available to build smart interfaces enabling more efficient applications, or at least to build other ways to exploit shop floor equipment. In fact, actual manufacturing systems have a lot of flexibility inside because they rely on programmable equipment, like robots and PLCs, to implement their functions. System engineers need to find ways to explore that flexibility when designing manufacturing systems, exposing it to the advanced user in more efficient ways.

In terms of operational results, it is important that a system like the one presented here does not add any production delay to the manufacturing system, or become a production bottleneck. This means that the cycle time should be lower than the cycle time of the previous station. In our example, the system takes around 11 seconds to perform the labeling operation, which is at least 20 seconds lower than the previous roll wrapping operation.

3.7.6 Conclusion

In describing an industrial application designed for labeling applications, this section discussed and detailed a software interface designed to command shop floor manufacturing systems remotely from the manufacturing tracking software. This interface added flexibility and agility to the manufacturing system, since all available operations were implemented in a very general way requiring only simple parameterization to specify the individual operations. The interface to the system is a simple TCP/IP server installed in one of the shop floor computers. To command the system, the client needs to connect to the server and, if allowed, send properly parameterized commands as simple messages over the open socket. The server uses *SUN RPC 4.0* compatible sockets to access the robotic system, place the received commands, and monitor the system operation. Since the TCP/IP server is a general implementation, using the BSD compatible TCP/IP socket implementation from *Microsoft*, it can receive commands from virtually any client. This makes the presented robotic cell interface an interesting way to command shop floor manufacturing systems.

3.8 References

[1] Halsall F., "Data Communications, Computer Networks and Open Systems", Third Edition, Addison-Wesley, 1992.

[2] Bloomer J., "Power Programming with RPC", O'Reilly & Associates, Inc., 1992.

[3] Siemens, "S7-200 System and Programming Manual", Siemens Automation, 2003.

[4] Mahalik, NP, "FieldBus Technology, Industrial Network Standards for Real-Time Distributed Control", Springer, 2003.

[5] Pires, JN, and Loureiro, Altino et al, "Welding Robots", IEEE Robotics and Automation Magazine, June, 2003

[6] Pires, JN, "Using Matlab to Interface Industrial Robotic & Automation Equipment", IEEE Robotics and Automation Magazine, September 2000

[7] Pires, JN, Sá da Costa, JMG, "Object-Oriented and Distributed Approach for Programming Robotic Manufacturing Cells", IFAC Journal Robotics and Computer Integrated Manufacturing, Volume 16, Number 1, pp. 29-42, March 2000.

[8] ABB Robotics, "RAP Service Specification, ABB Robotics, 2000.

[9] ABB Robotics, "S4C+ documentation CD" and "IRC5 documentation CD", ABB Robotics, 2000 and 2005

[10] ABB IRB140, IRB1400, IRB1400 & IRB6400 System Manual, ABB Robotics, Vasteras, Sweden, 2005.

[11] Visual Studio.NET 2003 Programmers Reference, Microsoft, 2003 (reference can be found at Microsoft's web site in the Visual C++ .NET location)

[12] Visual Studio.NET 2005 Programmers Reference, Microsoft, 2005 (reference can be found at Microsoft's web site in the Visual C++ .NET location)

4

Interface Devices and Systems

4.1 Introduction

The success of using robots with flexible manufacturing systems especially designed for small and medium enterprises (SME) depends on the human-machine interfaces (HMI) and on the operator skills. In fact, although many of these manufacturing systems are semi-autonomous, requiring only minor parameterization to work, many other systems working in SMEs require heavy parameterization and reconfiguration to adapt to the type of production that changes drastically with time and product models. Another difficulty is the average skill of the available operators, who usually have difficulty adapting to robotic and/or computer-controlled, flexible manufacturing systems.

SMEs are special types of companies. In dimension (with up to 250 permanent collaborators), in economic strength (with net sales up to 50M€) and in installed technical expertise (not many engineers). Nevertheless, the European economy depends on these types of company units since roughly they represent 95% of the European companies, more than 75% of the employment, and more than 60% of the overall net sales [1]. This reality configures a scenario in which flexible automation, and robotics in particular, play a special and unique role requiring manufacturing cells to be easily used by regular non-skilled operators, and easier to program, control and monitor. One way to this end is the exploitation of the consumer market's input-output devices to operate with industrial robotic equipment. With this approach, developers can benefit from the availability, and functionality of these devices, and from the powerful programming packages available for the most common desktop and embedded platforms. On the other hand, users could benefit from the operational gains obtained by having the normal tasks performed using common devices, and also from the reduction in prices due to the use of consumer products.

Industrial manufacturing systems would benefit greatly from improved interaction devices for human-machine interface even if the technology is not so advanced. Gains in autonomy, efficiency, and agility would be evident. The modern world requires better products at lower prices, requiring even more efficient manufacturing plants because the focus is on achieving better quality products, using faster and cheaper procedures. This means having systems that require less operator intervention to work normally, better human-machine interfaces, and cooperation between humans and machines sharing the same workspace as real coworkers.

Also, the robot and robotic cell programming task would benefit very much from improved and easy-to-use interaction devices. This means that availability of SDKs and programming libraries supported under common programming environments is necessary. Application development depends on that.

Working on future SMEs means considering humans and machines as coworkers, in environments where humans have constant access to the manufacturing equipment and related control systems.

Several devices are available for the user interface (several types of mice, joysticks, gamepads and controls, digital pens, pocket PCs and personal assistants, cameras, different types of sensors, etc.) with very nice characteristics that make them good candidates for industrial use. Integrating these devices with current industrial equipment requires the development of a device interface, which exhibits some basic principles in terms of software, hardware and interface to commercial controllers.

This scenario can be optimized in the following concurrent ways:

1. Develop user-friendly and highly graphical HMI applications to run on the available interface devices. Those environments tend to hide the complexity of the system from operators, allowing them to focus on controlling and operating the system. Figure 4.1 shows the main window of an application used to analyze force/torque data coming from a robotic system that uses a force/torque sensor to adjust the programmed trajectories (this system will not be further explored in this book)

2. Explore the utilization of consumer input/output devices that could be used to facilitate operator access to the system. In fact, there is a considerable amount of different devices on the market developed for personal computers on different input/output tasks. Such devices are usually programmable, with the manufacturers providing suitable SDKs to make them suitable for integrating with industrial manufacturing systems. Figure 4.2 shows a few of these devices, some of them covered in this book

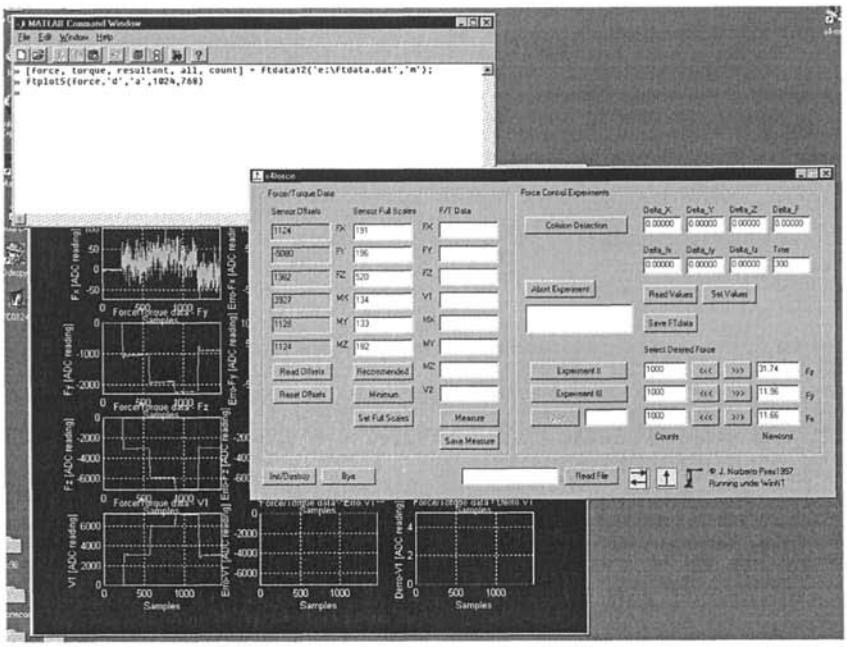


Figure 4.1 HMI interface used with an industrial robotic system to further analyze force/torque sensor data

3 Explore the functionality of the available software packages commonly used for engineering. Good examples of those packages are the CAD packages used by engineers to develop, optimize, and improve their designs (Figure 4.3). Since the vast majority of companies use CAD software packages to design their products, it would be very interesting if the information from CAD files could be used to generate robot programs. That is, the CAD application could be the environment used for specifying how robots should execute the required operations on the specified parts. Furthermore, since most engineers are familiar with CAD packages, exploring CAD data for robot programming and parameterization seems a good way to proceed [2].

Figure 4.2 Input/output devices used for HMI applications: (from top to bottom) joystick, headset with noise reduction, pocket PC and digital pen

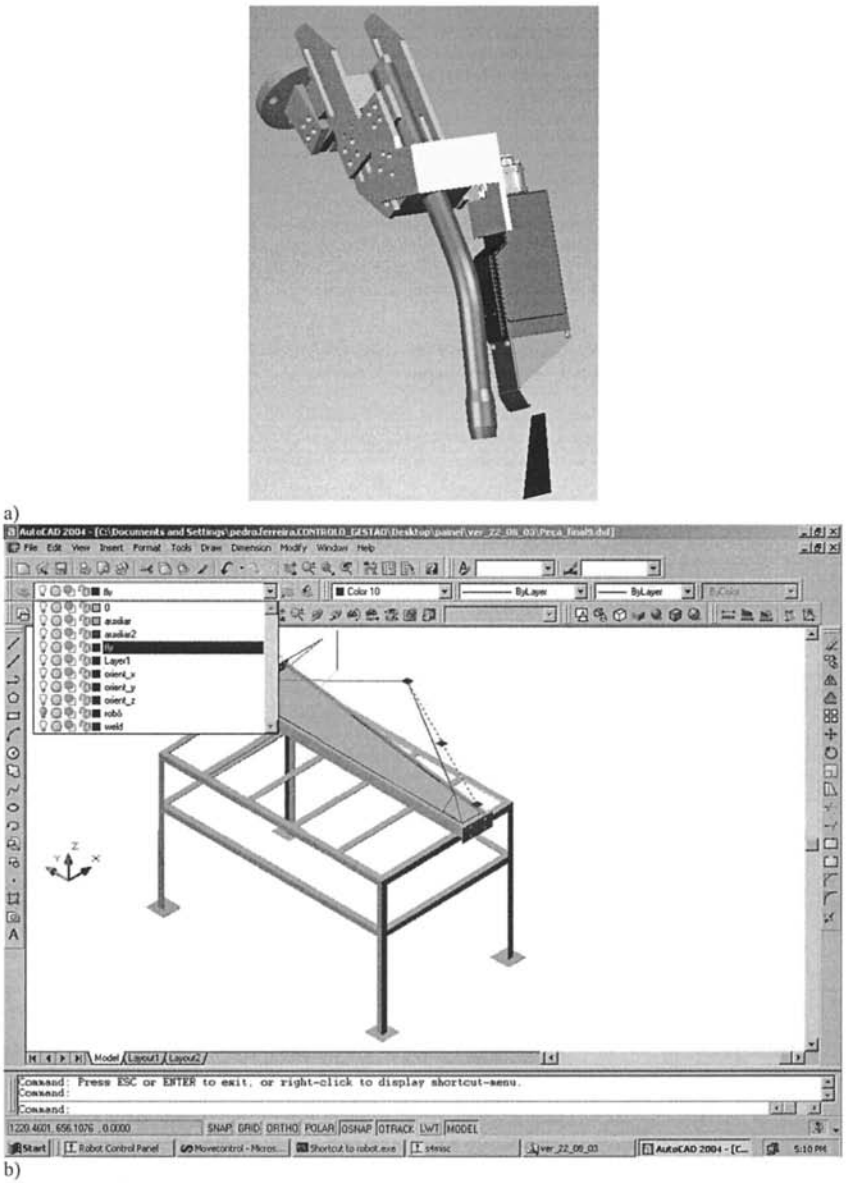


Figure 4.3 Using 3D CAD software packages to project and design mechanical parts: a – welding torch and laser camera (*SolidWorks*); b – welding trajectories specified using *AutoCad*

This chapter uses industrial and laboratory test-cases to provide the necessary details and insight to complement the above presented claims and design options.

4.2 Speech Interfaces

4.2.1 Introduction

Talking to machines is a thing normally associated with science fiction movies and cartoons and less with current industrial manufacturing systems. In fact, most of the papers about speech recognition start with something related to artificial intelligence, a science fiction movie, or a robot used in a movie, etc., where machines talk like humans, and understand the complex human speech without problems. Nevertheless, industrial manufacturing systems would benefit very much from speech recognition for human-machine interface (HMI) even if the technology is not so advanced. Gains in terms of autonomy, efficiency and agility seem evident. The modern world requires better products at lower prices, requiring even more efficient manufacturing plants because the focus is in achieving better quality products, using faster and cheaper procedures. This means autonomy, having systems that require less operator intervention to operate normally, better human-machine interfaces and cooperation between humans and machines sharing the same workspace as real coworkers.

The final objective is to achieve, in some cases, semi-autonomous systems [3], i.e., highly automated systems that require only minor operator intervention. In many industries, production is closed tracked in any part of the manufacturing cycle, which is composed by several in-line manufacturing systems that perform the necessary operations, transforming the raw materials in a final product. In many cases, if properly designed, those individual manufacturing systems require simple parameterization to execute the tasks they are designed to execute. If that parameterization can be commanded remotely by automatic means from where it is available, then the system becomes almost autonomous in the sense that operator intervention is reduced to the minimum and essentially related with small adjustments, error and maintenance situations [3]. In other cases, a close cooperation between humans and machines is desirable although very difficult to achieve, due to limitations of the actual robotic and automation systems.

The above described scenario puts focus on HMI, where speech interfaces play an important role because manufacturing system efficiency will increase if the interface is more natural or similar to how humans command things. Nevertheless, speech recognition is not a common feature among industrial applications, because:

- The speech recognition and text-to-speech technologies are relatively new, although they are already robust enough to be used with industrial applications
- The industrial environment is very noisy which puts enormous strain on automatic speech recognition systems
- Industrial systems weren't designed to incorporate these types of features, and usually don't have powerful computers dedicated to HMI

Automatic speech recognition (ASR) is commonly described as converting speech to text. The reverse process, in which text is converted to speech (TTS), is known as *speech synthesis*. Speech synthesizers often produce results that are not very natural sounding. Speech synthesis is different from voice processing, which involves digitizing, compressing (not always), recording, and then playing back snippets of speech. Voice processing results are natural sounding, but the technology is limited in flexibility and needs more disk storage space compared to speech synthesis.

Speech recognition developers are still searching for the perfect human-machine interface, a recognition engine that understands any speaker, interprets natural speech patterns, remains impervious to background noise, and has an infinite vocabulary with contextual understanding. However, practical product designers, OEMs, and VARs can indeed use today's speech recognition engines to make major improvements to today's markets and applications. Selecting such an engine for any product requires understanding how the speech technologies impact performance and cost factors, and how these factors fit in with the intended application.

Using speech interfaces is a big improvement to HMI systems, because of the following reasons:

- Speech is a natural interface, similar to the "*interface*" we share with other humans, that is robust enough to be used with demanding applications. That will change drastically the way humans interface with machines
- Speech makes robot control and supervision possible from simple multi-robot interfaces. In the presented cases, common PCs were used, along with a normal noise-suppressing headset microphone
- Speech reduces the amount and complexity of different HMI interfaces, usually developed for each application. Since a PC platform is used, which carry currently very good computing power, ASR systems become affordable and simple to use

In this section, an automatic speech recognition system is selected and used for the purpose of commanding a generic industrial manufacturing cell. The concepts are explained in detail and two test case examples are presented in a way to show that if certain measures are taken, ASR can be used with great success even with industrial applications. Noise is still a problem, but using a short command structure with a specific word as pre-command string it is possible to enormously reduce the noise effects. The system presented here uses this strategy and was tested with a simple noiseless pick-and-place example, but also with a simple welding application in which considerable noise is present.

4.2.2 Evolution

As already mentioned, the next level is to combine ASR with natural language understanding, i.e., making machines understand our complex language, coping with the implementations, and providing contextual understanding. That capability would make robots accessible to people who don't want to learn the technical details of using them. And that is really the aim, since a common operator does not have the time or the immediate interest to dig into technical details, which is, in fact, neither required nor an advantage.

Speech recognition has been integrated in several products currently available:

- Telephony applications
- Embedded systems (Telephone voice dialing system, car kits, PDAs, home automation systems, general use electronic appliances, etc.)
- Multimedia applications, like language learning tools
- Service robotics

Speech recognition has about 75 years of development. Mechanical devices to achieve speech synthesis were first devised in the early 19th century, but imagined and conceived for fiction stories much earlier.

The idea of an artificial speaker is very old, an aspect of the human long-standing fascination with humanoid *automata*. *Gerbert* (d. 1003), *Albertus Magnus* (1198-1280), and *Roger Bacon* (1214-1294) are all said to have built speaking heads. However, historically attested speech synthesis begins with *Wolfgang von Kempelen* (1734-1804), who published his findings of twenty years of research in 1791. *Wolfgang* ideas gain another interest with the invention of the telephone in the late 19th century, and the subsequent efforts to reduce the bandwidth requirements of transmitting voice.

On March 10, 1876, the telephone was born when *Alexander Graham Bell* called to his assistant, "*Mr. Watson! Come here! I want you!*" He was not simply making the first phone call. He was creating a revolution in communications and commerce. It started an era of instantaneous information-sharing across towns and continents (on a planetary level) and greatly accelerated economic development.

In 1922, a sound-activated toy dog named "*Rex*" (from *Elmwood Button Co.*) could be called by name from his doghouse.

In 1936, *U.K. Tel* introduced a "*speaking clock*" to tell time. In the 1930s, the telephone engineers at *Bell Labs* developed the famous *Voder*, a speech synthesizer that was unveiled to the public at the 1939 World's Fair, but that required a skilled human operator to operate with it.

Small vocabulary recognition was demonstrated for digits over the telephone by *Bell Labs* in 1952. The system used a very simple frequency splitter to generate

plots of the first two formants. The identification was achieved by matching them with a pre-stored pattern. With training, the recognition accuracy of spoken digits was 97%.

Fully automatic speech synthesis came in the early 1960s, with the invention of new automatic coding schemes, such as *adaptive predictive coding* (APC). With those new techniques in hand, the *Bell Labs* engineers again turned their attention to speech synthesis. By the late 1960s, they had developed a system for internal use in the telephone system, a machine that read wiring instructions to *Western Electric* telephone wirers, who could then keep eyes and hands on their work.

At the *Seattle World's Fair* in 1962, IBM demonstrated the "*Shoebox*" speech recognizer. The recognizer was able to understand 16 words (digits plus command/control words) interfaced with a mechanical calculator for performing arithmetic computations by voice. Based on mathematical modeling and optimization techniques learned at IDA (now the *Center for Communications Research*, Princeton), *Jim Baker* introduced stochastic processing with *hidden markov models* (HMM) to speech recognition while at *Carnegie-Mellon University* in 1972. At the same time, *Fred Jelinek*, coming from a background of information theory, independently developed HMM techniques for speech recognition at IBM. HMM provides a powerful mathematical tool for finding the invariant information in the speech signal. Over the next 10-15 years, as other laboratories gradually tested, understood, and applied this methodology, it became the dominant speech recognition methodology. Recent performance improvements have been achieved through the incorporation of discriminative training (at *Cambridge University*, LIMSI, etc.) and large databases for training.

Starting in the 1970s, government funding agencies throughout the world (e.g. *Alvey*, *ATR*, *DARPA*, *Esprit*, etc.) began making a major impact on expanding and directing speech technology for strategic purposes. These efforts have resulted in significant advances, especially for speech recognition, and have created large widely-available databases in many languages while fostering rigorous comparative testing and evaluation methodologies.

In the mid-1970s, small vocabulary commercial recognizers utilizing expensive custom hardware were introduced by *Threshold Technology* and *NEC*, primarily for hands-free industrial applications. In the late 1970s, *Verbex* (division *of Exxon Enterprises*), also using custom special-purpose hardware systems, was commercializing small vocabulary applications over the telephone, primarily for telephone toll management and financial services (e.g. Fidelity fund inquiries). By the mid-1990s, as computers became progressively more powerful, even large vocabulary speech recognition applications progressed from requiring hardware assists to being mainly based on software. As performance and capabilities increased, prices dropped.

Further progress led to the introduction, in 1976, of the *Kurzweil Reading Machine,* which, for the first time allowed the blind to "*read*" plain text as opposed

to *Braille*. By 1978, the technology was so well established and inexpensive to produce that it could be introduced in a toy, *Texas Instruments Speak-and-Spell*. Consequently, the development of this important technology from inception until fruition took about 15 years, involved practitioners from various disciplines, and had a far-reaching impact on other technologies and, through them, society as a whole.

Although existing for at least as long as speech synthesis, *automatic speech recognition* (ASR) has a shorter history. It needed much more the developments of *digital signal processing* (DSP) theory and techniques of the 1960s, such as *adaptive predictive coding* (APC), to even come under consideration for development.

Work in the early 1970s was again driven by the telephone industry, which hoped for both voice-activated dialing and also for security procedures based on voice recognition. Through gradual development in the 1980s and into the 1990s, error rates in both these areas were brought down to the point where the technologies could be commercialized.

In 1990, *Dragon Systems* (created by *Jim* and *Janet Bailer*) introduced a general-purpose discrete dictation system (i.e. requiring pauses between each spoken word), and in 1997, *Dragon* started shipping general purpose continuous speech dictation systems to allow any user to speak naturally to their computer instead of, or in addition to, typing. *IBM* rapidly followed the developments, as did *Lernout & Hauspie* (using technology acquired *from Kurzweil Applied Intelligence*), *Philips*, and more recently, *Microsoft*. Medical reporting and legal dictation are two of the largest market segments for ASR technology. Although intended for use by typical PC users, this technology has proven especially valuable to disabled and physically impaired users, including many who suffer from *repetitive stress injuries* (RSI). Robotics is also a very promising area.

AT&T introduced its automated operator system in 1992. In 1996, the company *Nuance* supplied recognition technology to allow customers of *Charles Schwab* to get stock quotes and to engage in financial transactions over the telephone. Similar recognition applications were also supplied by *SpeechWorks*. Today, it is possible to book airline reservations with *British Airways*, make a train reservation for *Amtrak*, and obtain weather forecasts and telephone directory information, all by using speech recognition technology. In 1997, *Apple Computer* introduced software for taking voice dictation in Mandarin Chinese.

Other important speech technologies include speaker verification/identification and spoken language learning for both literacy and interactive foreign language instruction. For information search and retrieval applications (e.g. audio mining) by voice, large vocabulary recognition preprocessing has proven highly effective, preserving acoustic as well as statistical semantic/syntactic information. This approach also has broad applications for speaker identification, language identification, and so on.

Today, 65 years after the *Voder* and just 45 years after APC, both ASR and TTS technologies can be said to be fully operational, in a case where a very convoluted technological history has had a modest and more or less anticipated social impact.

4.2.3 Technology

Speech recognition systems can be separated into several different classes depending on the types of utterances they have the ability to recognize. These classes are based on the fact that one of the difficulties of ASR is the ability to determine when a speaker starts and finishes an utterance. Most packages can fit into more than one class, depending on which mode they're using.

Isolated words: Isolated word recognizers usually require each utterance to have quiet (lack of an audio signal) on both sides of the sample window. It doesn't mean that it accepts single words, but does require a single utterance at a time. Often, these systems have "*listen/not-listen*" states, where they require the speaker to wait between utterances (usually doing processing during the pauses). Isolated utterance might be a better name for this class.

Connected words: Connected word systems (or more correctly "connected utterances") are similar to isolated words, but allow separate utterances to be run-together with a minimal pause between them.

Continuous speech: Continuous recognition is the next step. Recognizers with continuous speech capabilities are some of the most difficult to create because they must utilize special methods to determine utterance boundaries. Continuous speech recognizers allow users to speak almost naturally, while the computer determines the content. Basically, it's computer dictation and commanding.

Spontaneous speech: There appears to be a variety of definitions for what spontaneous speech actually is. At a basic level, it can be thought of as speech that is natural sounding and not rehearsed. An ASR system with spontaneous speech ability should be able to handle a variety of natural speech features such as words being run together, pauses, "ums" and "ahs", slight stutters, etc.

Voice verification/identification: Some ASR systems have the ability to identify specific users. This book doesn't cover verification or security systems, because user validation is done using other means.

Speech recognition, or speech-to-text, involves capturing and digitizing the sound waves, converting them to basic language units or phonemes, constructing words from phonemes, and contextually analyzing the words to ensure correct spelling for words that sound alike (such as "*write*" and "*right*").

Recognizers (also referred to as speech recognition engines) are the software drivers that convert the acoustic signal to a digital signal and deliver recognized speech as text to the application. Most recognizers support continuous speech, meaning the user can speak naturally into a microphone at the speed of most conversations. Isolated or discrete speech recognizers require the user to pause after each word, and are currently being replaced by continuous speech engines.

Continuous speech recognition engines currently support two modes of speech recognition:

- ***Dictation***, in which the user enters data by reading directly to the computer
- ***Command and control***, in which the user initiates actions by speaking commands or asking questions

Dictation mode allows users to dictate memos, letters, and e-mail messages, as well as to enter data using a speech recognition dictation engine. The possibilities for what can be recognized are limited by the size of the recognizer's "*grammar*" or dictionary of words. Most recognizers that support dictation mode are speaker-dependent, meaning that accuracy varies based on the user's speaking patterns and accent. To ensure accurate recognition, the application must create or access a "*speaker profile*" that includes a detailed map of the user's speech patterns captured in the matching process during recognition.

Command and control mode offers developers the easiest implementation of a speech interface in an existing application. In command and control mode, the grammar (or list of recognized words) can be limited to the list of available commands (a much more finite scope than that of continuous dictation grammars, which must encompass nearly the entire dictionary). This mode provides better accuracy and performance, and reduces the processing overhead required by the application. The limited grammar also enables speaker-independent processing, eliminating the need for speaker profiles or "*training*" the recognizer.

The **command and control mode** is the one most adapted for speech commanding of robots.

4.2.4 Automatic Speech Recognition System and Strategy

From the several continuous speech ASR technologies available, based on personal computers, the *Microsoft Speech Engine* [4] was selected because it integrates very well with the operating systems we use for HMI, manufacturing cell control, and supervision (*Windows XP/NT/2000*). The *Microsoft Speech Application Programming Interface* (SAPI) was also selected, along with the *Microsoft's Speech SDK* (version 5.1), to develop the speech and text-to-speech software applications [4]. This API provides a nice collection of methods and data structures that integrate very well in the *.NET 2003* framework [5], providing an interesting

developing platform that takes advantage of the computing power available from actual personal computers. Finally, the *Microsoft's SAPI 5.1* works with several ASR engines, which gives some freedom to developers to choose the technology and the speech engine.

Grammars define the way the ASR recognizes speech from the user. When a sequence included in the grammar is recognized, the engine originates an event that can be handled by the application to perform the planned actions. The SAPI provides the necessary methods and data structures to extract the relevant information from the generated event, so that proper identification and details are obtained.

There are three ways to define grammars: using XML files, using binary configuration files (CFG), or using the grammar builder methods and data structures. XML files are a good way to define grammars if a compiler and converter is available, as in the SDK 5.1. In the examples provided in this chapter, the grammar builder methods were used to programmatically construct and modify the grammar.

The strategy used here takes into consideration that there should be several robots in the network, running different applications. In that *scenario*, the user needs to identify the robot first, before sending the command. The following strategy is used,

- All commands start with the word "*Robot*"
- The second word identifies the robot by a number: one, two, etc
- The words that follow constitute the command and the parameters associated with a specific command

Consequently, the grammar used is composed of a "*TopLevelRule*" with a predetermined *initial state*, i.e., the ASR system looks for the pre-command word "*Robot*" as a precondition to any recognizable command string. The above mentioned sequence of words constitutes the *second level rules*, i.e, they are used by the *TopLevelRule* and aren't directly recognizable. A rule is defined for each planned action. As a result, the following represents the defined syntax of commands:

robot number command parameter_i

where "*robot*" is the pre-command word, *number* represents the robot number, *command* is the word representing the command to send to the robot, and *parameter_i* are *i* words representing the parameters associated with the *command*.

Another thing considered was safety. Each robot responds to "*hello*" commands, and when asked to "*initialize*" the robots require voice identification of username and password to give the user the proper access rights. Since the robots are connected to the calling PC using an RPC socket [2, 6-7] mechanism, the user must

"*initialize*" the robot to start using its remote services, which means that an RPC connection is open, and must "*terminate*" the connection when no more actions are needed. A typical session would look like,

User: Robot one hello.
Robot: I am listening my friend.
User: Robot one initialize.
Robot: You need to identify to access my functions.
Robot: Your username please?
User: Robot one *<username>*.
Robot: Correct.
Robot: Your password please?
User: Robot one *<password>*.
Robot: Correct.
Robot: Welcome again *<username>*. I am robot one. Long time no see.

Sequence of commands here. Robot is under user control.

User: Robot one terminate.
Robot: See you soon *<username>*.

In the following sections, two simple examples are given to demonstrate how this voice command mechanism is implemented, and how the robot controller software is designed to allow these features.

4.2.5 Pick-and-Place and Robotic Welding Examples

The following examples take advantage of developments done in the *Industrial Robotics Laboratory,* of the *Mechanical Engineering Department* of the *University of Coimbra* on robot remote access for command and supervision [2, 6-7]. Briefly, two industrial robots connected to an Ethernet network are used. The robot controllers (ABB S4CPlus) are equipped with RPC servers that enable user access from the network, offering several interesting services like variable access, IO access, program and file access and system status services [7]. The new versions of the ABB controller, named IRC5, are equipped with a TCP/IP sockets API [8], enabling users to program and setup TCP/IP sockets servers in the controller. For that reason, the ideas presented here can be easily transported to the new IRC5 controller with no major change.

If calls to those services are implemented in the client PC, it is fairly easy to develop new services. The examples presented here include the *ActiveX PCROBNET2003* [9] that implement the necessary methods and data structures (see Table 3.3) to access all the services available from the robot controller.

The basic idea is simple and not very different from the concept used when implementing any remote server. If the system designer can access robot program variables, then he can design his own services and offer them to the remote clients. A simple *SWITCH-CASE-DO* cycle, driven by a variable controlled from the calling client, would do the job:

```
switch (decision_1)
{
  case 0: call service_0; break;
  case 1: call service_1; break;
  case 2: call service_2; break;
  ...
  case n: call service_n; break;
}
```

4.2.6 Pick-and-Place Example

For example, consider a simple pick-and-place application. The robot, equipped with a two-finger pneumatic gripper, is able to pick a piece from one position (named "*origin*") and deliver it to other position (named "*final*"). Both positions are placed on top of a working table (Figure 4.4).

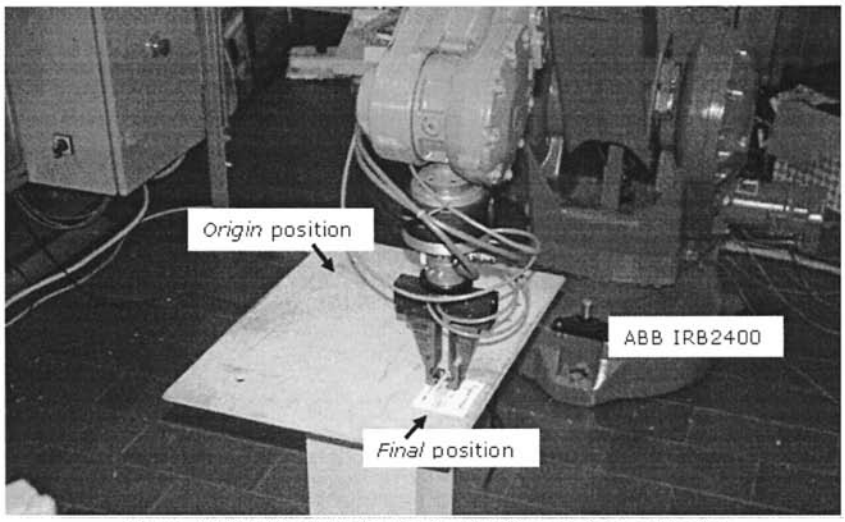


Figure 4.4 Working table for the simple pick-and-place application

The robot can be commanded to open/close the gripper, approach origin/final position (positions 100mm above origin/final position, respectively), move to origin/final position, and move to "*home*" (a safe position away from the table). This is a simple example, but sufficient to demonstrate the voice interface. Figure

4.5 shows a simplified version of the server software running on the robot controller.

To be able to send any of those commands using the human voice, the following grammar was implemented:

TopLevelRule = "*Robot*"	pre-command word
Rule 0 = "*one hello*"	check if robot is there
Rule 1 = "*one initialize*"	ask robot to initialize (*open client*)
Rule 2 = "*one master*"	rule defining username "*master*"
Rule 3 = "*one masterxyz*"	password of username "*master*"
Rule 4 = "*one open*"	open the gripper
Rule 5 = "*one close*"	close the gripper
Rule 6 = "*one motor on*"	put robot in run state
Rule 7 = "*one motor off*"	put robot in stand-by state
Rule 8 = "*one program run*"	start program
Rule 9 = "*one program stop*"	stop program
Rule 10 = "*one approach origin*"	call service 94
Rule 11 = "*one approach final*"	call service 93
Rule 12 = "*one origin*"	call service 91
Rule 13 = "*one final*"	call service 92
Rule 14 = "*one home*"	call service 90
Rule 15 = "*one terminate*"	release robot access (close client)

```
PROC main()
  TPErase; TPWrite "Example Server ...";
  p1:=CRobT(\Tool:=trj_tool\WObj:=trj_wobj);
  MoveJ p1,v100,fine,trj_tool\WObj:=trj_wobj;
  decision1:=123;
  WHILE TRUE DO
    TEST decision1
      CASE 90:
        MoveJ home,v200,fine,tool0; decision1:=123;
      CASE 91:
        MoveL final,v200,fine,tool0; decision1:=123;
      CASE 92:
        MoveL origin,v200,fine,tool0; decision1:=123;
      CASE 93:
        MoveJ Offs(final, 0,0,100),v200,fine,tool0; decision1:=123;
      CASE 94:
        MoveJ Offs(origin, 0,0,100),v200,fine,tool0; decision1:=123;
    ENDTEST
  ENDWHILE
ENDPROC
```

Figure 4.5 Simple pick-and-place server implemented in RAPID

The presented rules were introduced into a new grammar using the grammar builder included in the *Microsoft Speech API* (SAPI) [4]. The following (Figure 4.6) shows how that can be done, using the *Microsoft Visual Basic .NET2003* compiler.

```
TopRule = Grammar.Rules.Add("TopLevelRule",
SpeechLib.SpeechRuleAttributes.SRATopLevel Or
SpeechLib.SpeechRuleAttributes.SRADynamic, 1)

ListItemsRule = Grammar.Rules.Add("ListItemsRule",
SpeechLib.SpeechRuleAttributes.SRADynamic, 2)

AfterCmdState = TopRule.AddState
m_PreCommandString = "Robot"
TopRule.InitialState.AddWordTransition(AfterCmdState, m_PreCommandString, " ", , "",
0, 0)

AfterCmdState.AddRuleTransition(Nothing, ListItemsRule, "", 1, 1)
ListItemsRule.Clear()

ListItemsRule.InitialState.AddWordTransition(Nothing, "one hello", " ", , "one hello", 0, 0)
...
Grammar.Rules.Commit()
Grammar.CmdSetRuleState("TopLevelRule",SpeechLib.SpeechRuleState.SGDSActive)
RecoContext.State() = SpeechLib.SpeechRecoContextState.SRCS_Enabled
```

Figure 4.6 Adding grammar rules and compiling the grammar using SAPI in *Visual Basic .NET2003*

After committing and activating the grammar, the ASR listens for voice commands and generates speech recognition events when a programmed command is recognized. The corresponding event service routines execute the commanded strings. Figure 4.7 shows the shell of the application built in *Visual Basic .NET 2003* to implement the voice interface for this simple example. Two robots are listed in the interface. The robot executing the simple pick-and-place example is robot one (named *Rita*).

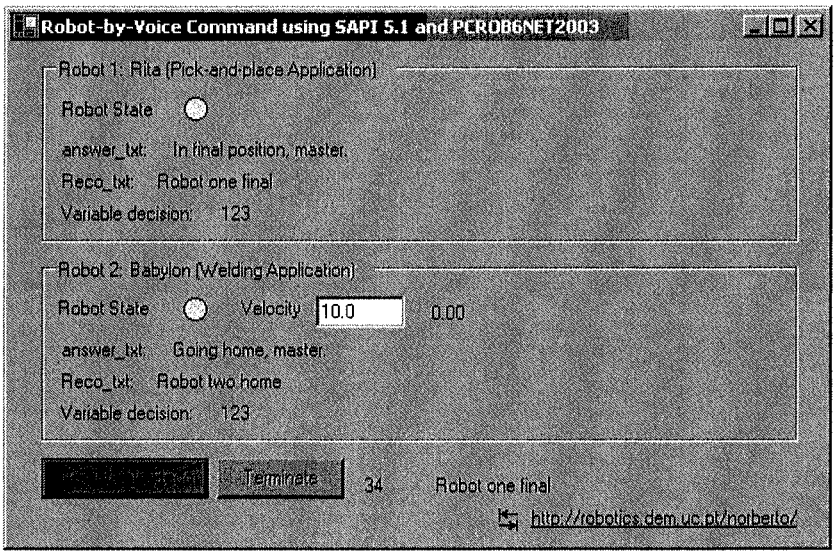


Figure 4.7 Shell of the voice interface application used to command the robot

With this interface activated, the following sequence of commands (admitting that the logging procedure was already executed) will take the robot from the "*home*" position, pick the work object at the origin position, deliver it to the final position, return to "*home*" and release the robot control.

User: Robot one approach origin.
Robot: Near origin, master.
User: Robot one open.
Robot: Tool open master.
User: Robot one origin.
Robot: In origin position master.
User: Robot one close.
Robot: Tool close master.
User: Robot one approach origin.
Robot: Near origin, master.
User: Robot one approach final.
Robot: Near final, master.
User: Robot one final.
Robot: In final position, master.
User: Robot one approach final.
Robot: Near final, master.
User: Robot one home.
Robot: In home position, master.
User: Robot one terminate.
Robot: See you soon master.

The speech event routine, running on the voice interface application, is called when any of the rules defined in the working grammar are recognized. For example, when the "*motor on*" rule is identified, the following routine is executed:

```
If ok_command_1 = 1 And (strText = "Robot one motor on") Then
   result1 = Pcrobnet2003.MotorON2(1)
   If result1 >= 0 Then
      Voice.Speak("Motor on, master.")
      ans_robot_1.Text() = "Motor ON, master."
   Else
      Voice.Speak("Error executing, master.")
      ans_robot_1.Text() = "Error executing, master."
   End If
End If
```

To give another example, when the move to "origin" rule is recognized, the following routine is executed:

```
If ok_command_1 = 1 And (strText = "Robot one origin") Then
   Dim valor As Integer
   valor = 92
   result1 = Pcrobnet2003.WriteNum2("decision1", valor, 1)
   If result1 >= 0 Then
      Voice.Speak("In origin position, master.")
      ans_robot_1.Text() = "In origin position, master."
   Else
      Voice.Speak("Error executing, master.")
      ans_robot_1.Text() = "Error executing, master."
   End If
End If
```

4.2.7 Robotic Welding Example

The welding example presented here extends slightly the functionality of the simple server presented in Figure 4.5, just by adding another service and the necessary routines to control the welding power source. The system used for this demonstration is composed of an industrial robot ABB IRB1400 equipped with the robot controller *ABB S4CPlus*, and a MIG/MAG welding power source (*ESAB LUA 315A*). The work-piece is placed on top of a welding table, and the robot must approach point 1 (named "*origin*") and perform a linear weld from that point until point 2 (named "*final*"). The system is presented in Figure 4.8. The user is able to command the robot to

- Approach and reach the point origin (P1)
- Approach and reach the point final (P2)

- Move to "*home*" position
- Perform a linear weld from point P1 (origin) to point P2 (final)
- Adjust and read the value of the welding velocity

These actions are only demonstration actions selected to show further details about the voice interface to industrial robots. To implement the simple welding server, it is enough to add the following welding service to the simple server presented in Figure 4.5:

```
CASE 94:
  weld_on;
  MoveL final,v200,fine,tool0;
  weld_off;
  decision1:=123;
```

where the routine "*weld_on*" makes the necessary actions to initiate the welding arc [2], and the routine "*weld_off*" performs the post welding actions to finish the welding and terminate the welding arc [2].

The welding server is running in robot 2 (named *babylon*), and is addressed by that number from the voice interface application (Figure 4.9). To execute a linear weld from P1 to P2, at 10mm/s, the user must command the following actions (after logging to access the robot, and editing the velocity value in the voice interface application – Figure 4.9) using the human voice:

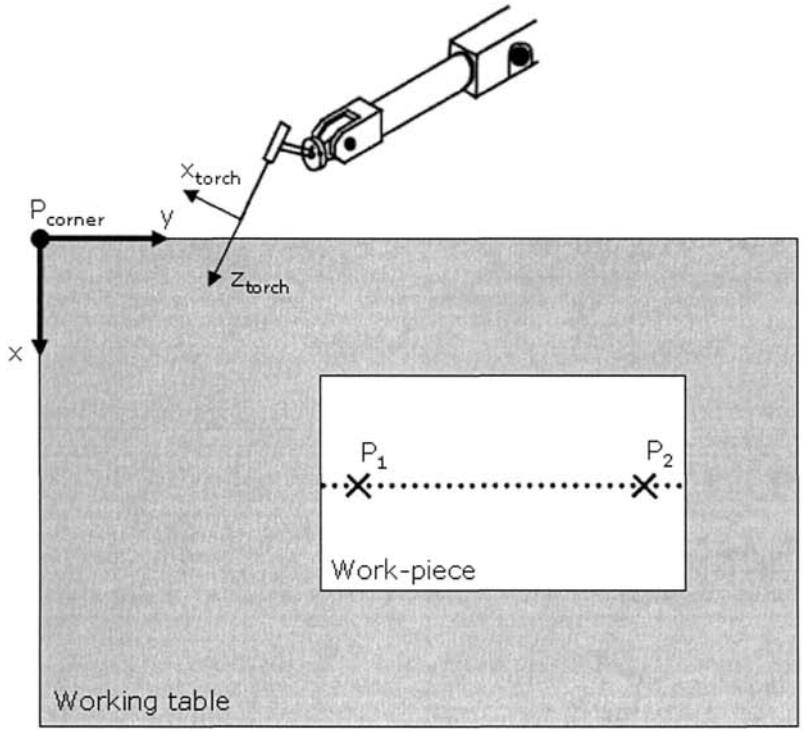
User: Robot two approach origin.
Robot: Near origin master.
User: Robot two origin.
Robot: In origin position master.
User: Robot two velocity.
Robot: Velocity changed master.
User: Robot two weld.
Robot: I am welding master.
User: Robot two approach final.
Robot: Near final master.

Figure 4.9 shows the voice interface when robot two is actually welding along with a user equipped with a handset microphone to send voice commands to the robot. The code associated with the welding command is,

```
If ok_command_2 = 1 And (strText = "Robot two weld") Then
  Dim valor As Integer
  valor = 95
  result1 = Pcrobnet2003.WriteNum2("decision1", valor, 2)
  If result1 >= 0 Then
    Voice.Speak("I am welding, master.")
    ans_robot_2.Text() = "I am welding, master."
```

```
    Else
      Voice.Speak("Error executing, master.")
      ans_robot_2.Text() = "Error executing, master."
    End If
  End If
```

The code above writes the value 95 to the variable "*decision1*", which means that the service "*weld*" is executed (check Figure 4.5).



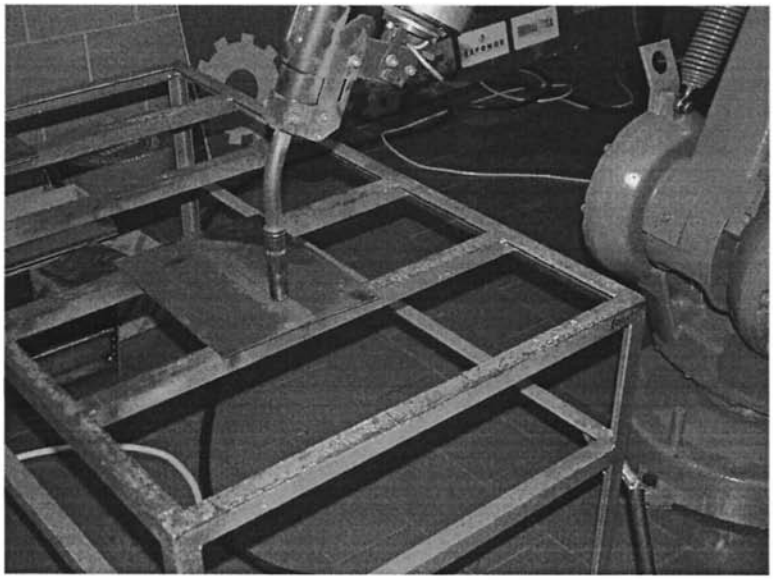


Figure 4.8 Simple welding application used for demonstration

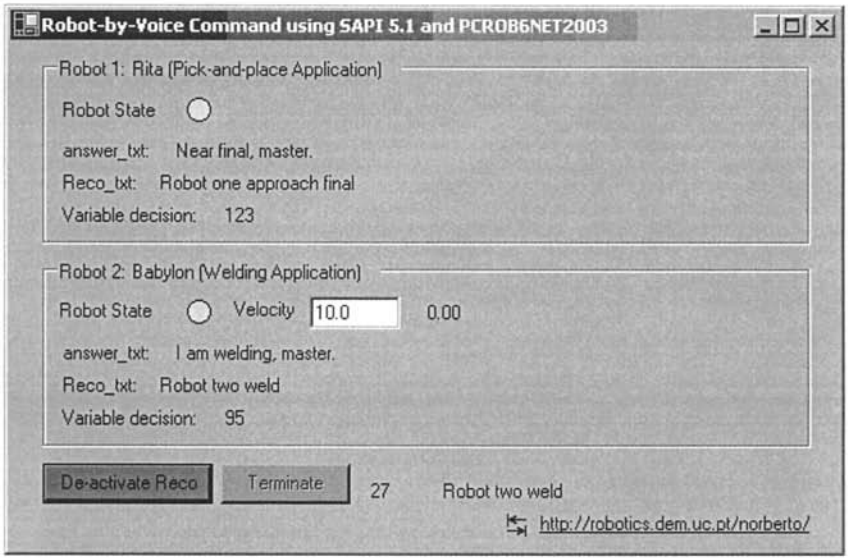


Figure 4.9 Shell of the voice interface application showing the welding operation, and a user (author of this book) commanding the robot using a headset microphone

4.2.8 Adjusting Process Variables

During the welding process, it may be necessary to adjust process variables such as the welding velocity, welding current, the welding points, and so on. This means that the voice interface must allow users to command numerical values that are difficult to recognize with high accuracy. Furthermore, it is not practical to define fixed rules for each possible number to recognize, which means that dictation capabilities must be active when the user wants to command numbers. To avoid noise effects, and consequently erroneous recognition, a set of rules were added to enable dictation only when necessary, having the rule strategy defined above always active. Consequently, the following rules were added for robot two (the one executing the welding example):

Rule V1 = "two variables"	enables access to variables
Rule V2 = "two variables out"	ends access to variables
Rule V3 = "two <*variable_name*>"	enables access to <*variable_name*>
Rule V4 = "two <*variable_name*> lock"	ends access to <*variable_name*>
Rule V5 = "two <*variable_name*> read"	reads from <*variable_name*>

Rule V6 = "two <*variable_name*> write" writes to <*variable_name*>

Rules V1 and V2 are used to activate/deactivate the dictation capabilities, which will enable the easy recognition of numbers in decimal format (when the feature is activated, a white dot appears in the program shell – Figure 4.10). Rules V3 and V4 are used to access a specific variable. When activated, each number correctly recognized is added to the text box associated with the variable (a blinking LED appears in the program shell – Figure 4.10). Deactivating the access, the value is locked and can be written to the robot program variable under consideration. The rules V5 and V6 are used to read/write the actual value of the selected variable from/to the robot controller.

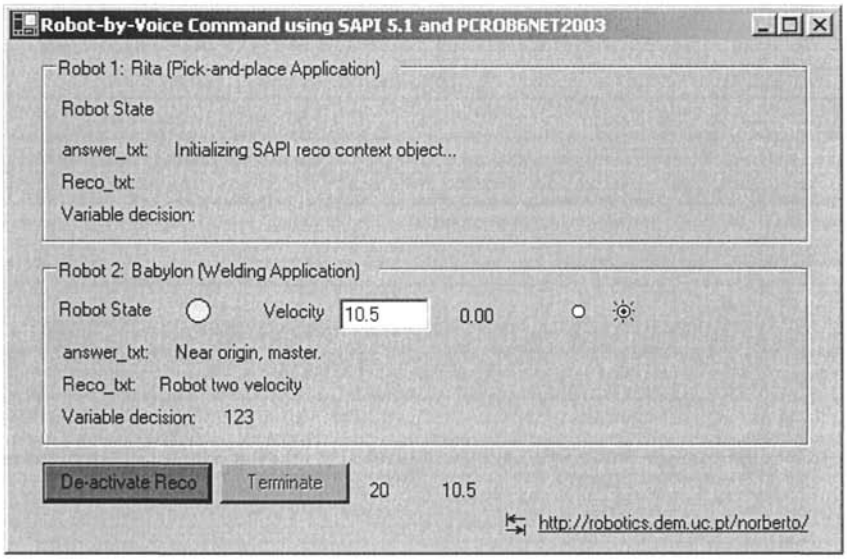


Figure 4.10 Accessing variables in the robot controller

As an example, to adjust the welding velocity the following code is executed after the corresponding rule is recognized:

```
If ok_command_2 = 1 And (strText = "Robot two velocity write") Then
   Dim valor as Double
   Dim velocity as Integer
   valor = velocity.Text()
   result1 = Pcrobnet2003.WriteSpeed("velocity", valor, 2)
   If Result11 >= 0 Then
      Voice.Speak("Welding velocity changed, master.")
      ans_robot_2.Text() = "Welding velocity changed, master."
   Else
      Voice.Speak("Error executing, master.")
```

```
        ans_robot_2.Text() = "Error executing, master."
    End If
End If
```

Because the voice interface was designed to operate with several robots, two in the present case, the user may send commands to both robots using the same interface which is potentially interesting.

Using speech interfaces is a big improvement to HMI systems, for the following reasons:

- Speech is a natural interface, similar to the "*interface*" we share with other humans, that is robust enough to be used with demanding applications. It will change drastically how humans interface with machines
- Speech makes robot control and supervision possible from simple multi-robot interfaces. In the presented cases, common PC's were used, along with a quite normal noise-suppressing headset microphone
- Speech reduces the amount and complexity of different HMI interfaces, usually developed for each application. Since a PC platform is used, and they carry very good computing power, ASR systems become affordable and user-friendly

The experiments performed with this interface worked extremely well, even when high noise was involved (namely during welding applications), which indicates clearly that the technology is suitable to use with industrial applications where human-machine cooperation is necessary or where operator intervention is minimal.

4.2.9 Conclusion

In this section, a voice interface to command robotic manufacturing cells was designed and presented. The speech recognition interface strategy used was briefly introduced and explained. Two selected industrial representative examples were presented to demonstrate the potential interest of these human-machine interfaces for industrial applications.

Details about implementation were presented to enable the reader to immediately explore from the discussed concepts and examples. Because a personal computer platform is used, along with standard programming tools (*Microsoft Visual Studio .NET2003* and *Speech SDK 5.1*) and an ASR system freely available (SAPI 5.1), the whole implementation is affordable even for SME utilization.

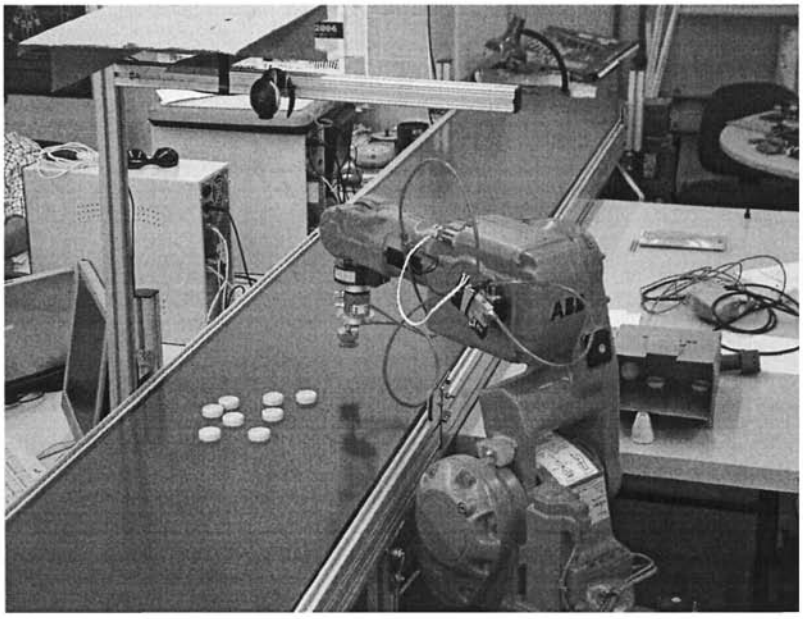
The presented code and examples, along with the fairly interesting and reliable results, indicate clearly that the technology is suitable for industrial utilization.

4.3 VoiceRobCam: Speech Interface for Robotics

The example presented in this section extends the example in section 3.2, namely adding extra equipment and implementing a simple manufacturing cell-like system composed of a robot, a conveyor, and several sensors. It also includes a voice/speech interface developed to allow the user to command the system using his voice. The reader should consider the presented example as a demonstration of functionality because many of the options were taken with that objective in mind, rather than trying to find the most efficient solutions but instead the ones that suit better the demonstrating purpose.

The system (Figure 4.11) used in this example is composed of:

- An industrial robot ABB IRB140 [8] equipped with the new IRC5 robot controller
- An industrial conveyor, fully equipped with presence sensors, and actuated by an electric AC motor managed through a frequency inverter. To control the conveyor, an industrial PLC (*Siemens S7-200*) [12] is used
- A Webcam used to acquire images from the working place and identify the number and position of the available objects. The image processing software runs on a PC offering remote services through a TCP/IP sockets server



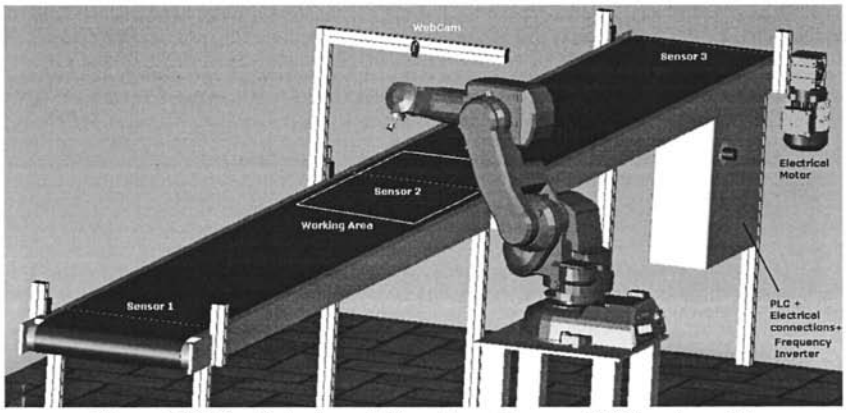


Figure 4.11 Manufacturing cell-like setup: picture and *Solidworks* model

In the following, a brief explanation of how the various subsystems work is provided. In the process, the relevant details about each subsystem and their respective construction are also given.

4.3.1 Robot Manipulator and Robot Controller

The ABB IRB140 (Figure 4.12) is an anthropomorphic industrial robot manipulator designed to be used with applications that require high precision and repeatability on a reduced working place. Examples of those types of applications are welding, assembly, deburring, handling, and packing.

ABB IRB 140 Basic Details

Year of release: 1999
Repeatability: +/- 0.03mm
Payload: 5kg
Reach: 810mm
Max. TCP Velocity: 2.5m/s
Max. TCP Acceleration: 20m/s2
Acceleration time 0-1m/s: 0.15 seconds

Figure 4.12 Details about the industrial robot manipulator ABB IRB140

This robot is equipped with the new IRC5 robot controller from *ABB Robotics* (Figure 4.13). This controller provides outstanding robot control capabilities, programming environment and features, along with advanced system and human machine interfaces.

IRC5 Basic Details

Year of release: 2005
Multitask system
Multiprocessor system
Powerful programming language: RAPID
FieldBus scanners: Can, DeviceNet, ProfiBus, Interbus
DeviNet Gateway: Allen-Bradley remote IO
Interfaces: Ethernet, COM ports
Protocols: TCP/IP, FTP, Sockets
Pendant: WinCE based teach-pendant
PLC-like capabilities for IO

Figure 4.13 Details about the industrial robot controller IRC5

The robot is programmed in this application to operate in the same way as explained in section 3.3.1, i.e., a TCP/IP socket server is available that offers services to the remote clients (see Table 3.3). This server is independent of the particular task designed for the robot, and allows only the remote user to send commands and influence the running task. In this case, the task is basically to pick objects from the conveyor and place them on a box. The robot receives complete commands specifying the position of the object to pick. Furthermore, since the relevant robot IO signals are connected to the PLC, the robot *status* and any IO action, like "*MOTOR ON/OFF*", "*PROGRAM RUN/STOP*", "*EMERGENCY*", etc., are obtained through the PLC interface.

4.3.2 PLC Siemens S7-200 and Server

The PLC (Figure 4.14) plays a central role in this application, as it is common in a typical industrial manufacturing setup where the task of managing the cell is generally done by a PLC. In this example, to operate with the PLC, a server was developed to enable users to request PLC actions and to obtain information from the setup. To make the interface simple and efficient, the server accepts TCP/IP socket connections, offering the necessary services to the client's applications. The list of available services is presented in Table 4.1. The client application just needs to connect to the PLC server software application to be able to control the setup and obtain status and process information.

The server application (Figure 4.15) runs on a computer that is connected to the PLC through the RS232C serial port, and to the local area network (LAN) for client access.

Table 4.1 Services available from the PLC TCP/IP server

Service	Answer	Description
Init_Auto	<Init_Auto	Conveyor in Automatic Mode
Init_Manual	<Init_Auto	Conveyor in Manual Mode
Stop	<Stop>	Conveyor in STOP Mode
Read_Mode	Auto, Manual e Stop	Returns the conveyor operating mode
Manual_Forward	Manual_Forward	Conveyor starts in Manual Mode
Manual_Stop	Manual_Stop	Conveyor stops in Manual Mode
Force_Forward	<Force_Forward	Forces the conveyor to Start, although in Automatic Mode
IO	Bit stream*	Returns the status of all IO signals
Status	Bit stream**	Returns the status of all IO signals and the conveyor operating mode
Motor_On	<Motor_On>	Robot Motor ON
Motor_Off	<Motor_Off>	Robot Motor OFF
Prg_Run	<Prg_Run>	Robot Program RUN
Prg_Stop	<Prg_Stop>	Robot Program STOP

* The IO bit stream is formated in the following format:

BQ0.0:xxxxxxxxBQ1.0:xxxxxxxx BI0.0:xxxxxxxx:BI1.0:xxxxxxxx

where "*BQ0.0:*"/"*BI0.0:*" is string followed by 8 bits corresponding to the first block of digital outputs/inputs of the PLC, "*BQ1.0:*"/"*BI1.0:*" is a string followed by the 8 bits corresponding to the second block of digital outputs/inputs. For example, the following answer is obtained when BQ0.2, BQ1.0, BQ1.4, BQ1.6, BI0.1, BI1.0, BI1.1 and BI1.2 are activated:

BQ0.0:00100000BQ1.0:10001010BI0.0:01000000:BI1.0:11100000

Note: The bit assignment is as follows:

BQ0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
	Conv_F	Conv_B	user	M_on	user	P_run	P_stop	M_off
BQ1	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
	user	user	user	user	user	user	user	User
BI0*	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
	Auto	Manual	M_on	M_off	P_run	P_stop	EMS	Busy
BI1	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
	Sensor1	Sensor2	Sensor3	User	user	user	user	user

*BI0 contains robot status information as listed.

** Similar to the above bit stream, but with the string "*Auto*", "*Manual*", or "*Stop*" added in the end of the stream in accordance with the state of the conveyor. For example, for the above mentioned IO state and with the conveyor in *Automatic Mode*, the answer to the *Status* call is,

BQ0.0:00100000BQ1.0:10001010BI0.0:01000000:BI1.0:11100000_Auto

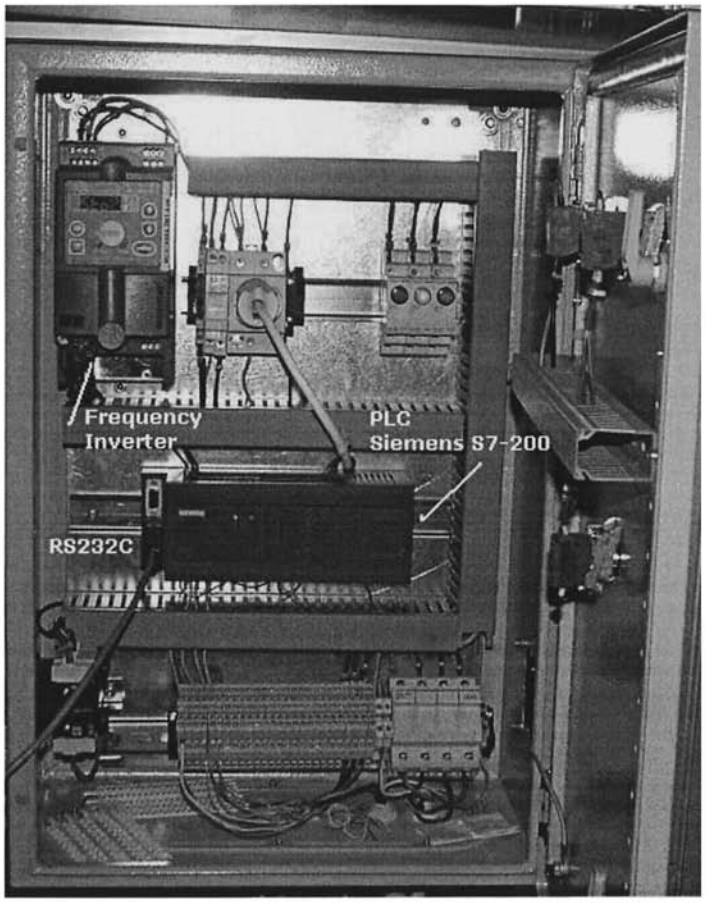


Figure 4.14 Electrical panel showing the PLC, the frequency inverter and the electrical connections

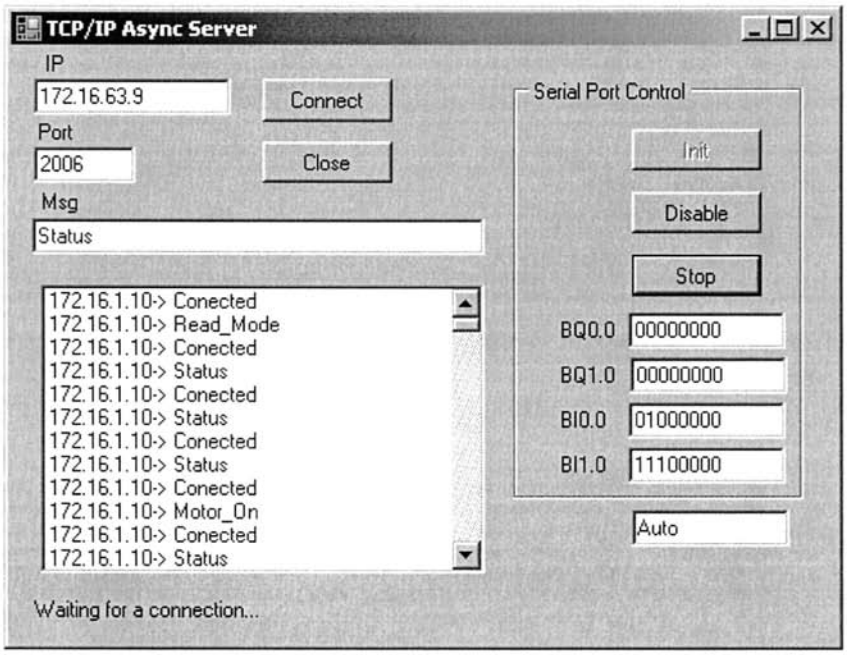


Figure 4.15 Shell of the PLC TCP/IP socket server

The PLC works as a server, as explained in Section 3.2.1.2, offering the IO services and actions necessary to control the system and obtain status information.

4.3.3 Webcam and Image Processing Software

This setup uses a simple USB *Webcam* to obtain images from the working area and compute the number of objects present and their respective positions. The camera is connected to a PC that runs the image processing software developed in *LabView* from *National Instruments* using the *IMAQ Vision toolbox*. The software works in the same way as explained in Section 3.3.2. Nevertheless, two more messages were added to the TCP/IP server, which return's the information necessary to calibrate the camera and to compute the object position in the robot's cartesian space (Table 4.2).

Table 4.2 Services from the Webcam TCP/IP server

Service	Description
camera get objects	Gets a frame from the *Webcam*
calibration pixels	Correlation between pixels and millimeters
cam to pos X_Y	Offset to add to the (x, y) position obtained from the image to compute the position of the object in the robot Cartesian space

The image processing software waits for a "*camera acquire objects*" message from the user client. When a message arrives, the server acquires a frame (image) from the camera and performs a binary operation, i.e., from a color image, or with several levels of gray, a back-and-white image is obtained with only two colors: black (0) or white (1). This type of procedure is necessary to identify the working objects in the scene and remove the unnecessary light effects.

The next task is to remove all the objects that are out of the working range. Those correspond to the parts of the conveyor belt, light effects, shadows, etc., and need to be removed before identifying the objects and computing their position.

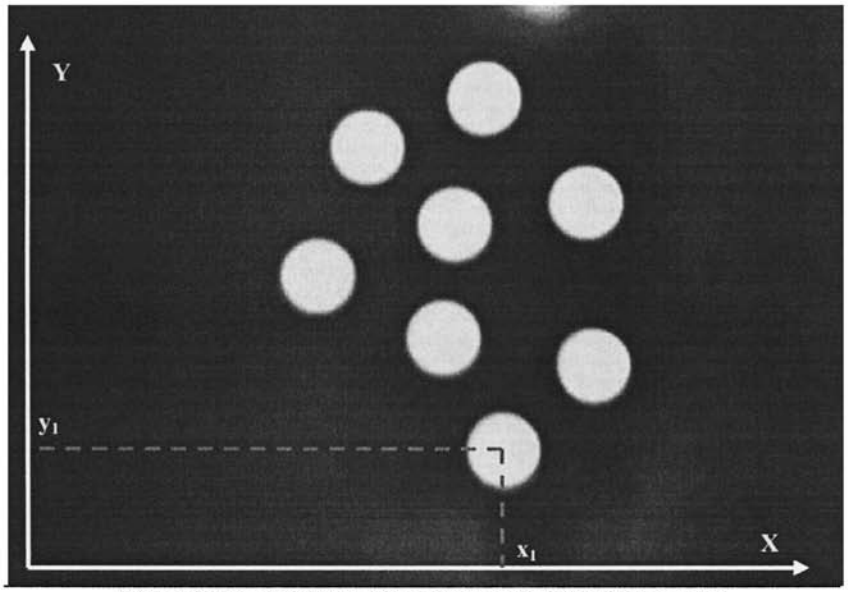


Figure 4.16 Frame obtained from the camera after being processed

Because the objects used with this application are small discs without holes (Figure 4.16), the image processing software uses a procedure to fill all the holes resulting from the binary operation. After that, a valid object should have a number of pixels between certain limits. This will allow users to identify unknown objects or objects that are overlapped. Only objects that pass this identification are considered, and for those the center of mass is computed: All other objects are ignored. From that the (x, y) position is computed and returned to the client application that issued the call.

4.3.4 User Client Application

It is now easy to understand the software architecture designed for this application (Figure 4.17): distributed and based on a client-server model. The user client application just need's to implement calls to the available services, track the answers, and monitor the robot and conveyor operations with the objective of controlling the setup.

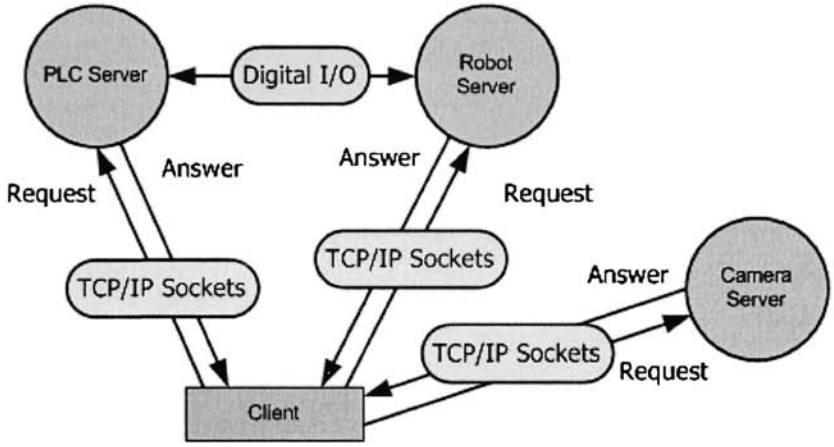
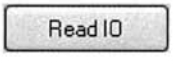


Figure 4.17 Basic distributed software architecture and connections between the different software modules

Figure 4.18 shows the shell of a PC client application developed using *C# .NET 2005* to access the above mentioned TCP/IP services from the various servers, and control the manufacturing cell-like system. With this application, the user can operate the setup in "*Manual Mode*", i.e., issue all the actions independently, and at a time. The user can also have the conveyor in "*Automatic Mode*" and command the pick-and-place operation manually, i.e., require "*camera get objects*" to obtain the number of objects and their respective positions, selecting from the obtained list of objects the ones to pick.

Finally, the user can command the setup to work in fully "*Automatic Mode*", i.e., to start the conveyor when objects are detected by sensor 1 (Figure 4.11), stop the conveyor when objects are detected by sensor 2, acquire an image of the working space and identify the number of objects and their positions, and then pick-and-place all of them and resume the conveyor operation.

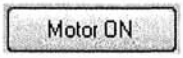
For example, the "*Read IO*" and "*Motor ON*" actions are implemented with the following code:



Send the message "IO" to the PLC TCP/IP socket server and process the returned answer

```
private void bt_ReadIO_Click(object sender, EventArgs e)
{
  int rec_num; string str_temp;
  m_socClient1 = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
  IPEndPoint remoteEP_PLC = new IPEndPoint(IPAddress.Parse("172.16.63.9"), 2006);
  m_socClient1.Connect(remoteEP_PLC);
  m_socClient1.Send(System.Text.Encoding.ASCII.GetBytes("IO<E>"));
  byte[] recData = new byte[256];
  rec_num = m_socClient1.Receive(recData);
  m_socClient1.Close();
  if (recData[6] == 48)
  {
    tapete = false;
    Tapete_ON.Checked = false;
  }
  else
  {
    conveyor = true; conveyor_ON.Checked = true;
  }
  if (recData[34] == 48)
  {
    sensor1 = false; sensor1.Checked = false;
  }
  else
  {
    sensor1 = true; sensor1.Checked = true;
  }
  if (recData[35] == 48)
  {
     sensor2 = false; sensor2.Checked = false;
  }
  else
  {
    sensor2 = true; sensor2.Checked = true;
  }
  if (recData[36] == 48)
  {
     sensor3 = false; sensor3.Checked = false;
  }
  else
  {
    sensor3 = true; sensor3.Checked = true;
  }
  str_temp = System.Text.Encoding.ASCII.GetString(recData, 0, rec_num);
}
```

Presenting the received information



Send the message that commands the robot "*Motor ON*" action

```
private void bt_Motor_ON_Click(object sender, EventArgs e)
{
  m_socClient1 = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
  IPEndPoint remoteEP_PLC = new IPEndPoint(IPAddress.Parse("172.16.63.9"), 2006);
  m_socClient1.Connect(remoteEP_PLC);
  m_socClient1.Send(System.Text.Encoding.ASCII.GetBytes("Motor_On<E>"));
  byte[] recData = new byte[256];

  m_socClient1.Receive(recData);
  m_socClient1.Close();
}
```

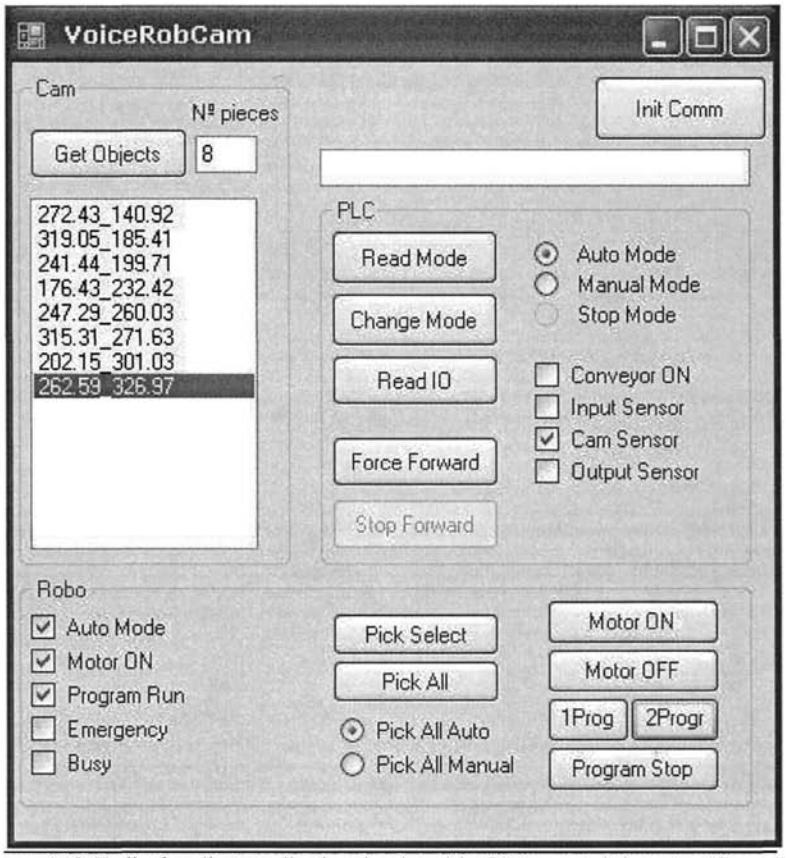


Figure 4.18 Shell of a client application developed in C# to control the setup (Sensor1 = "*Input Sensor*", Sensor2 = "*Cam Sensor*" and Sensor3 = "*Output Sensor*")

The client code is very simple and is composed of five main parts:

- Established socket client connection
- Send the command message
- Receive the answer
- Process and present the returned information
- Close the socket

When operating in fully "*Automatic Mode*" follows the sequence represented in Figure 4.19, which corresponds to the normal (or production-like) operation of the system. Considering a real production setup, it could be interesting to have more portable solutions. Consequently, a client application (Figure 4.20) was also developed to run on a *Pocket PC* (PPC). This application has the same basic functionality of the PC application (Figure 4.18).

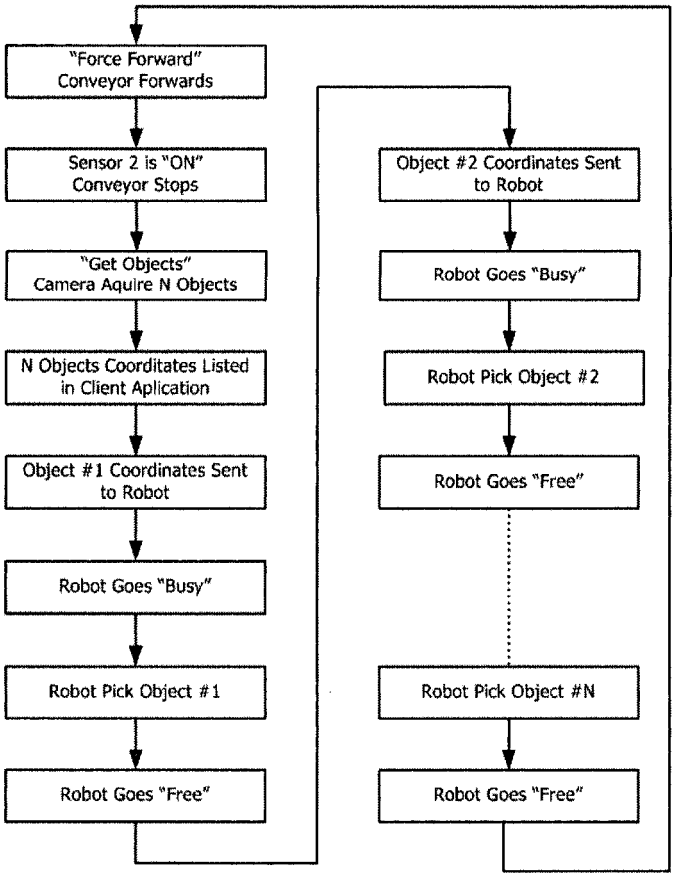


Figure 4.19 Sequence for the fully "*Automatic Mode*"

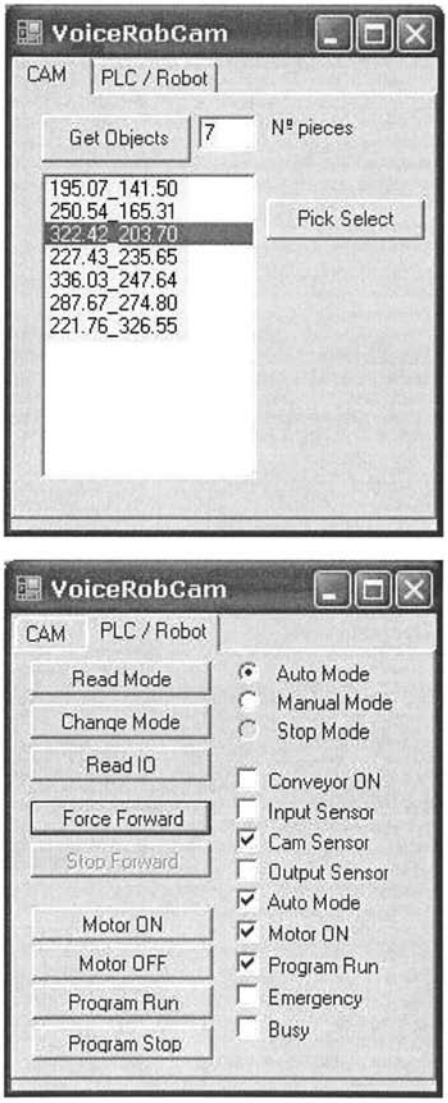


Figure 4.20 Aspects of the PPC client application developed in C# to control the manufacturing cell-like setup (Sensor1 = "*Input Sensor*", Sensor2 = "*Cam Sensor*" and Sensor3 = "*Output Sensor*")

4.3.5 Speech Interface

The current example is an interesting platform to demonstrate the potential of developing speech recognition systems for human-machine interfaces in industrial manufacturing systems. This statement is based on the following arguments:

- The system is constituted exclusively of industrial equipment, which makes it representative of a typical robotic manufacturing cell
- The software architecture developed to handle the system is distributed and based on a client-server model. This is a current trend in actual manufacturing plants
- The system uses industrial standards for network communications (Ethernet and TCP/IP)
- The system software was developed using commonly available software tools: *Microsoft Visual Studio .NET 2005*
- The concepts and technologies used in the system, for software, communications system organization, etc., are commonly accessible and most of them are currently defined as standards

As explained earlier, the system can be commanded manually, i.e., the various subsystems that compose the system can be directly commanded by the user. That perspective, or mode of operation, is explored in this section to introduce and demonstrate the enormous potential of current speech recognition (ASR) and text-to-speech (TTS) engines. In the presented implementation, the *Microsoft Speech API 5.1* (SAPI 5.1) is used to add speech recognition features (*speech commands*) to any of the above presented applications.

The strategy used to build the speech recognition grammar is simple and based on the concepts already presented in section 4.2. Since the system used here is composed of three different subsystems, a pre-command string per each piece of equipment is needed in the speech grammar. This allows the user to address each subsystem by its name,

m_PreCommandString1 = "Robot"
m_PreCommandString2 = "Conveyor"
m_PreCommandString3 = "Camera"

These three words ("*Robot*", "*Conveyor*", and "*Camera*") are added to the speech recognition grammar as *TopLevelRules*, i.e., those words need to be identified to start the recognition of a command string. This means that the speech recognition grammar is built considering that the user commands have the following structure:

name_of_subsystem command parameters

where "*name_of_subsystem*" is one of the *TopLevelRules*, i.e., one of the words that identify each of the subsystems, "*command*" is a string identifying the command, and "*parameters*" is a string containing the parameters associated with

the specified command. Consequently, to have the system responding to speech commands, it is necessary to first identify the commands of interest and their associated parameters (Table 4.3).

Table 4.3 Commands associated with the speech command interface

TopRule	***Robot***	
Command	**Parameters**	**Description**
Hello	--	Checks if the speech recognition system is ready
Initialize	--	Initializes the interface and starts the login procedure, requesting *username* and *password*
Terminate	--	Terminates the speech interface.
<username>	--	Validates the "*username*"
<password>	--	Validates the "*password*"
Motor	On	Robot in Motors On state
	Off	Robot in Motors Off state
Program	Run	Starts loaded program from the beginning
	Stop	Stops loaded program
	Run from point	Starts loaded program from the actual program point
Program Option	Root	Selects program option "*root*": start menu
	<Number>	Selects program option defined by "*number*"
Pick	<Number>	Pick object defined by "*number*"
TopRule	***Conveyor***	
Command	**Parameters**	**Description**
Auto	--	Places conveyor in Automatic Mode
	Start	Forces the conveyor to start moving
Manual	--	Places conveyor in Manual Mode
	Start	Conveyor starts moving
	Stop	Conveyor stops moving
TopRule	***Camera***	
Command	**Parameters**	**Description**
Get Objects	--	Returns the number of objects in the scene and their respective positions
Calibration Pixels		Returns the pixel to millimeters ratio
Cam to Pos X_Y		Returns the offset that should be added to the computed positions to obtain the position in the robot Cartesian space

Therefore, adding the above presented rules to the speech recognition grammar (using an XML file or directly in the code), the ASR mechanism fires events when a rule is correctly identified. Consequently, the client application should just track the ASR generated events, discriminate the rule that was identified, and execute the associated actions. To perform those tasks, the ASR API provides functions that return the identified rule as a string. The application just needs to compare the string with the relevant possibilities, activating the associated actions when a match is obtained. Figure 4.21 shows some detail about the code associated with adding a speech commanding interface to the current application. Only the relevant parts of the code are listed, taking, as example, a few selected functions.

Speech Recognition Event Routine

```
...
  strText = Result.PhraseInfo.GetText(0, -1, True)
...
  If ok_command_1 = 0 And (strText = "Robot initialize") Then
    Voice.Speak("Your Username please?")
    ans_robot_1.Text() = "Your Username please?"
    ok_command_1 = -1
    asr_state.Text() = "Username."
  End If
...
  If ok_command_1 = -1 And (strText = "Robot master") Then
    Voice.Speak("Correct. And your password please?")
    ans_robot_1.Text() = "Correct. And your password please?"
    ok_command_1 = -2
    asr_state.Text() = "Password."
  End If
...
  If ok_command_1 = -2 And (strText = "Robot access level three") Then
    Voice.Speak("Correct. Welcome again master. Long time no see.")
    ans_robot_1.Text() = "Correct. Welcome again, master. Long time no see."
    ok_command_1 = 1
    If (result1 >= 0) Then
      robot1_on.Visible() = True
      asr_state.Text() = "Login OK."
    End If
  End If
...
  If ok_command_1 = 1 And (strText = "Robot terminate") Then
    Voice.Speak("See you soon, master.")
    ans_robot_1.Text() = "See you soon, master."
    s.Close()
    ok_command_1 = 0
    If (robot1_on.Visible = True) Then
      robot1_on.Visible = False
      asr_state.Text() = "Logout."
```

```
  End If
End If

If ok_command_1 = 1 And (strText = "Robot motor on") Then
  s = ConnectSocket(server_name, server_port)
  If s Is Nothing Then
    ans_robot.Text() = "Error connecting to robot, master"
    Voice.Speak("Error connecting to robot, master")
  Else
    Dim bytesSent As [Byte]() = Nothing
    bytesSent = ascii.GetBytes("motor_on")
    s.Send(bytesSent, bytesSent.Length, 0)
    'Voice.Speak("Motor on command received, master.")
    bytes = s.Receive(bytesReceived, bytesReceived.Length, 0)
    If Encoding.ASCII.GetString(bytesReceived, 0, bytes) = "0" Then
      Voice.Speak("Motor on, master.")
      ans_robot_1.Text() = "Motor on, master."
    Else
      Voice.Speak("Error executing, master.")
      ans_robot_1.Text() = "Error executing, master."
    End If
  End If
End If

If ok_command_1 = 1 And (strText = "Robot pick eight") Then
  s = ConnectSocket(server_name, server_port)
  If s Is Nothing Then
    ans_robot.Text() = "Error connecting to robot, master"
    Voice.Speak("Error connecting to robot, master")
  Else
    Dim bytesSent As [Byte]() =
    ascii.GetBytes("command_str 5000_" + object_cam(8))
    s.Send(bytesSent, bytesSent.Length, 0)
    bytes = s.Receive(bytesReceived, bytesReceived.Length, 0)
    ans_robot.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
    's.Close()
    If Encoding.ASCII.GetString(bytesReceived, 0, bytes) = "0" Then
      Voice.Speak("Robot pick, master.")
      ans_robot_1.Text() = "Robot pick, master."
    Else
      Voice.Speak("Error executing, master.")
      ans_robot_1.Text() = "Error executing, master."
    End If
  End If
End If
```

```
If ok_command_1 = 1 And (strText = "Conveyor manual start") Then
   s = ConnectSocket(server_name_plc, server_port_plc)
   If s Is Nothing Then
      ans_robot.Text() = "Error connecting to conveyor, master"
      Voice.Speak("Error connecting to conveyor, master")
   Else
      Dim bytesSent As [Byte]() = ascii.GetBytes("Manual_Forward<E>")
      s.Send(bytesSent, bytesSent.Length, 0)
      bytes = s.Receive(bytesReceived, bytesReceived.Length, 0)
      pdata.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
      Voice.Speak("Conveyor manual start, master.")
      ans_robot_1.Text() = "Conveyor manual start, master."
   End If
End If

If ok_command_1 = 1 And (strText = "Conveyor auto start") Then
   s = ConnectSocket(server_name_plc, server_port_plc)
   If s Is Nothing Then
      ans_robot.Text() = "Error connecting to conveyor, master"
      Voice.Speak("Error connecting to conveyor, master")
   Else
      Dim bytesSent As [Byte]() = ascii.GetBytes("Force_Forward<E>")
      s.Send(bytesSent, bytesSent.Length, 0)
      bytes = s.Receive(bytesReceived, bytesReceived.Length, 0)
      pdata.Text() = Encoding.ASCII.GetString(bytesReceived, 0, bytes)
      Voice.Speak("Conveyor automatic start, master.")
      ans_robot_1.Text() = "Conveyor automatic start, master."
   End If
End If
```

Figure 4.21 Detail about the code used in the ASR event routine

With this type of procedure, it is fairly simple add speech recognition features to the client applications described in this section. In general terms, the following is necessary (or desirable) to use speech commanding with industrial manufacturing systems:

- The system must be distributed in terms of software and based on a client-server model
- All the necessary subsystems must implement some type of mechanism for remote access from remote clients: RPC, TCP/IP sockets, etc
- A clear definition of the commanding strings must be available for easy implementation in different environments
- The speech recognition grammar developed for the application must reflect the above definitions. The routines associated with the recognition events must implement the service calls (using the defined commanding strings) and process the answers

- Some type of access mechanism must be implemented for security and safety reasons
- Critical commands should require some type of confirmation to avoid damaging persons and parts
- A careful selection of the headset used to implement the speech interface must be done, namely selecting devices with noise reduction electronics and with a press-to-speak switch

With these basic guidelines, speech recognition can be successfully added to industrial systems, resulting in a speech-enabled human-machine interface that could be a valuable improvement in terms of operator adaptation to the system. This would then improve operator productivity and efficiency, which would then impact the overall competitiveness of the company.

4.4 CAD Interfaces

Since the vast majority of companies use CAD software packages to design their products, it would be very interesting if the information from CAD files could be used to generate robot welding programs. That is, the CAD application could be the environment used for specifying how the welding robots should execute the welding operation on the specified parts.

Furthermore, because most welding engineers are familiar with CAD packages, this could be a nice way to proceed. An application presented elsewhere [2, 13-14] enables the user to work on the CAD file, defining both the welding path and the approach/escape paths between two consecutive welds, and organize them into the desired welding sequence. When the definition is complete, a small program, written in *Visual Basic*, extracts motion information from the CAD file and converts it to robot commands that can be immediately tested for detailed tuning. A set of tools is then available to speed up the necessary corrections, which can be made online with the robot moving. After a few simulations (with the robot performing all the programmed motions without welding) the program is ready for production. The whole process can be completed in just some minutes to a few hours, depending on the size and complexity of the component to be welded, representing a huge reduction of programming and set up time. Besides, most of the work is really easy offline programming.

These issues are further researched elsewhere [2, 13-14]. The objective here is to focus on the CAD interface and on adding more functionality to the human-machine interface of welding robots. Here the parameterization approach will be used. With this approach, the welding information, extracted from the CAD model, is used to parameterize a generic existing robot program, *i.e.*, the welding routines are implemented as general as possible enabling the accommodation of the planned welding tasks. In the case presented here, the information extracted from the CAD file, and adjusted using the presented software tools, is stored in a "*.wdf*" file and

sent to the robot controller using the option "*Send to Robot*" of the software tool. The information is sent in the form of single column matrices serialized by the sequence that must be followed, *i.e.*, each line of any matrix contains the information corresponding to a certain welding point. As already mentioned, the robot controller is organized as a server, offering a collection of services to the remote computer. Therefore, the following are examples of services implemented in the welding server, running on a ABBIRB1400 industrial robot equipped with the S4C+ robot controller (the same robot used in Section 4.2).

Service 9100 (Move_CRobot): this service is used to move the robot in the Cartesian space with the specified TOOL frame, in accordance with the commanded offsets: x, y, z, rx, ry, and rz, where (x, y, z) are the Cartesian offsets and (rx, ry, rz) are the rotation offsets about the tool axis x, y and z, respectively.

Service 9401 (Welding): this service is used to execute the welding sequence commanded to the robot.

Service 9301 (Simulation): this service is used to execute the welding sequence without igniting the arc, *i.e.*, the welding power source is not activated.

Service 9101 (Move_JRobot): this service is used to move the robot in the joint space in accordance with absolute joint angles commanded from the remote computer.

Consequently, the main routine of the welding server may be implemented as a simple *SWITCH-CASE-DO* cycle, driven by a variable controlled from the remote computer (Figure 4.22).

Looking into the code in more detail, it's easy to find out how it works and how it can be explored, but also how new functions can be added to the system. Let's consider for example the *Move_CRobot* service (Figure 4.22) that corresponds to the value 9100 of the variable *decision1*. To move the robot in the cartesian space, the following must be commanded from the remote computer.

1. Enter the service routine: to do that, the user must write the value 9100 to the numeric variable *decision1*. The method from the *PCROBNET2003/2005* software component used to command that task is:

```
pcrob.WriteNum("decision1", 9100, channel);
```

where *channel* identifies the RPC socket open between the robot controller and the remote computer.

```
PROC main()
    TPErase; TPWrite "Welding Server ...";
    reset_signals;
    p1:=CRobT(\Tool:=trj_tool\WObj:=trj_wobj);
    MoveJ p1,v100,fine,trj_tool\WObj:=trj_wobj;
    joints_now:=CJointT();
    decision1:=123; varmove:=0;
    WHILE TRUE DO
      TEST decision1
        CASE 9100:
          x:=0; y:=0; z:=0; rx:=0; ry:=0; rz:=0; move:=0;
          p1:=CRobT(\Tool:=trj_tool);
          WHILE (decision1=9100) DO
            IF (move <> 0) THEN
              p1:=RelTool(p1,x,y,z\Rx:=rx\Ry:=ry\Rz:=rz);
              x:=0; y:=0; z:=0; rx:=0; ry:=0; rz:=0; move:=0;
            ENDIF
            IF varmove <> 198 THEN
              MoveJ p1,v100,fine,trj_tool\WObj:=trj_wobj;
            ELSE
              MoveL p1,v100,fine,trj_tool\WObj:=trj_wobj;
            ENDIF
          ENDWHILE
          decision1:=123; varmove:=0;
        CASE 9101:
          joints_now:=CJointT();
          WHILE decision1=9101 DO
            MoveAbsJ joints_now,v100,fine,trj_tool\WObj:=trj_wobj;
          ENDWHILE
          decision1:=123;
        CASE 9401:
          weld;
          decision1:=123;
          p1:=CRobT(\Tool:=trj_tool);
          MoveJ RelTool(p1,0,0,-200),v100,fine,trj_tool\WObj:=trj_wobj;
        CASE 9301:
          weld_sim;
          decision1:=123;
      ENDTEST
    ENDWHILE
ENDPROC
```

Figure 4.22 Simple welding server running on the robot controller

2. Define the type of motion: the user must specify what type of motion to perform to achieve the target point, *i.e.*, linear motion or coordinated joint motion. This is specified writing to the variable *varmove* (198 for joint coordinated motion and any other value for linear motion). For example, the command

```
pcrob.WriteNum("varmove", 198, channel);
```

specifies joint coordinated motion, using the open RPC socket identified by the parameter *channel*.

3. Command the Cartesian and rotational offsets: the user must write the offsets to the corresponding variables. After that, when the user signals that the offsets are available (writing a value different than zero to the variable *move*), the robot moves to the position/orientation obtained by adding those offsets to the actual position, and waits for another motion command. For example, the sequence of commands necessary to move the robot 20 mm in the positive X direction and 10 mm in the negative Z direction should be:

```
pcrob.WriteNum("x", 20, channel);
pcrob.WriteNum("y", -10, channel);
pcrob.WriteNum("move", 1, channel);   <------- robot moves now!
```

where again *channel* identifies the open RPC socket.

4. Leave the service: to leave this service the user must write any value different from 9100 to the variable *decision1*. For example, the following command writes the value -1 to the numeric variable *decision1* and makes the robot program quit the *Move_CRobot* service:

```
pcrob.WriteNum("decision1", -1, channel);
```

Finally, let's consider the service *Welding* (Figure 4.22) that corresponds to the value 9401 of the variable *decision1*. The simplified version of the code is presented in Figure 4.23.

It is clear from the presented code (Figure 4.23) that the user should command the *Welding* service to execute, after sending the matrices defining the welding sequence. This service commands the robot to follow exactly the command sequence, moving the robot and igniting or stopping the welding arc whenever in the presence of a welding or approach/escape trajectory, respectively.

The example shows clearly that there are considerable gains in terms of flexibility and agility when using distributed client-server software architecture to assist industrial welding operations [2], namely taking advantage of the powerful programming tools developed for personal computers. It also shows that actual CAD packages can be used for robot programming tasks with great advantage, which extend the interest of already largely utilized software tools.

```
PROC weld()
    weldon:=0; i:=1;
    WHILE ((decision1=9401) AND (i<=numberpoints) AND (i>=1)) DO
        weldpoint:=i;
        wd_iref:=trj_voltage{i}; feed_iref:=trj_current{i};
        wd_href:=trj_voltage{i}; feed_href:=trj_current{i};
        wd_ref:=trj_voltage{i}; feed_ref:=trj_current{i};
        IF (trj_type{i}=0) THEN
          weld_on;
          weldon:=1;
        ENDIF
        ppos:=trj{i}; pvel:=trj_vel{i};
        pzone:=trj_prec{i}; ptype:=trj_mode{i};
        move_gen;
        IF (weldon=1) AND ((i+1>numberpoints) OR (trj_type{i+1}=1)) THEN
          weld_off;
          weldon:=0;
        ENDIF
        i:=i+1;
    ENDWHILE
    IF (weldon=1) THEN
      weld_off;
      weldon:=0;
    ENDIF
ENDPROC

PROC move_gen()
    IF ptype=0 THEN
      MoveL ppos,pvel,pzone,trj_tool\WObj:=trj_wobj;
    ENDIF
    IF ptype=1 THEN
      MoveJ ppos,pvel,pzone,trj_tool\WObj:=trj_wobj;
    ENDIF
    IF ptype=2 THEN
      TPWrite "[MOVE_GEN]: MoveC not implemented.";
    ENDIF
ENDPROC
```

Welding definition

Move the robot

Figure 4.23 Code for the *Welding* service

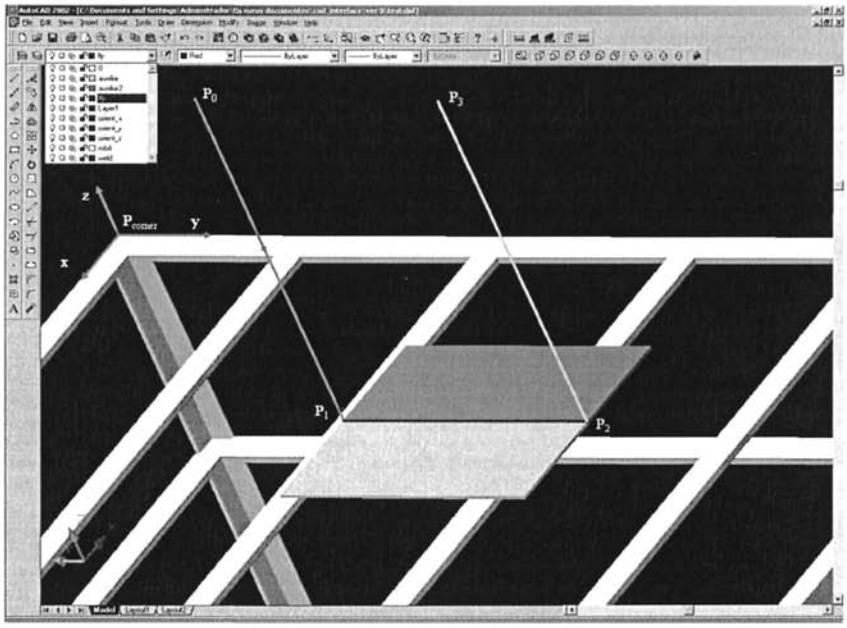


Figure 4.24 Definition of the simple welding example using *AUTOCAD*

To clarify further, let's consider the simple welding example already used in section 4.2.7. In that example, the robot is commanded to execute a linear welding on a work piece placed on a welding table. To demonstrate how this simple task is completely specified and programmed using a CAD package, the welding table and work piece were modeled in *AUTOCAD*. The same strategy used before is again utilized to specify points/orientations and trajectories, *i.e.*, they are all defined relative to a work object point/orientation (or reference system) named P_{corner}. In this way, when exporting points/orientations and trajectories to the robot, the only thing needed is a good calibration procedure of the robot TCP relative to P_{corner}, which can be done automatically using sensors (for example, laser position sensors) and special alignment routines, or manually using the robot joystick.

To execute the welding operation it is necessary to specify four points/orientations (P_0 to P_3) and the trajectories between them (Figure 4.24). The following procedures should be used:

1. P_0 should be defined as the approach point/orientation, *i.e.*, a point/orientation that could permit the robot to reach safely the work-piece from the "*home*" position. P_0 is consequently a non-welding point/orientation and the trajectory to P_0 should be free of obstacles (the user should guarantee adjusting P_0 accordingly). The precision to reach P_0 should be specified as low.

2. The trajectory from P_0 to P_1 should be defined as an approach linear trajectory, with point P_1 reached with the highest precision at low/medium velocity (let say 100mm/s, for example). As defined in [2], the weld layers in *AUTOCAD* are named for easy identification using a string that starts with the word "*WELD*". The next information is the type of trajectory, to distinguish between welding trajectories and approach/escape trajectories. After that should be specified the *welding current*, and then the *welding voltage*. Finally, the *welding speed* is specified. All these parameters are separated by spaces, constituting a definition string. Consequently, the label associated with that trajectory [2, 13-14] should be

   WELD 1 0 0 0 100 0

   for an approach/escape trajectory, done at 100mm/s with highest precision in the endpoint.

3. The trajectory from P_1 to P_2 should be defined as a welding trajectory with the required welding parameters. For example, the following label could be associated with this trajectory:

   WELD 0 150.0 21.3 10 0

   for a welding trajectory executed at 10mm/s, with highest precision in the end-point, associated with a welding current of 150.0 A and a welding voltage of 21.3 V.

4. The trajectory from P_2 to P_3 should be defined as an approach/escape trajectory done with low/medium velocity without any special precision in the endpoint. The following label could be associated with this trajectory:

   WELD 1 0 0 0 100 50

   to specify a trajectory done at 100mm/s, with low precision (50 mm sphere around the selected point).

This information is saved in the CAD file and can be extracted to a "*.wdf*" definition file, which is used for simulation and final tuning using the available tools [2, 13-14]. Finally, all of the information is sent to the robot using the already presented procedures, based on the routines developed for the robot controller and the "*write variable*" services (see Table 3.3) available from the *ActiveX* software [9] component used.

4.4.1 Speech Interface for Welding

Considering the linear weld case presented in Figure 4.24, a simple application was developed to command the welding procedure using a speech commanding

interface. This is particularly relevant because the welding cells are usually very noisy and not attractive to operators, namely the younger ones. Consider that the trajectories were planned in *AUTOCAD* and transferred to the robot using the above mentioned applications. To operate the robot, the speech commands presented in Table 4.4 are necessary.

Table 4.4 Speech commands for the simple welding application

TopLevelRule	***Robot***	***Number = Two***
Command	**Parameters**	**Description**
Hello	--	Checks if the speech recognition system is ready
Initialize	--	Initializes the interface and starts the login procedure, requesting *username* and *password.*
Terminate	--	Terminates the speech interface.
<username>	--	Validates the "*username*"
<password>	--	Validates the "*password*"
Motor	On	Robot in Motors On state
	Off	Robot in Motors Off state
Program	Run	Starts loaded program from the beginning
	Stop	Stops loaded program
	Run from point	Starts loaded program from the actual program point
Approach	Origin	Approach "*Origin*" position
	Final	Approach "*Final*" position
Origin	---	Move to "*Origin*" position
Final	---	Move "*Final*" position
Weld	---	Perform a weld operation from "*Origin*" position to "*Final*" position

Note: The command message was defined in sections 4.2.4 and 4.2.7.

The application presented in Figure 4.10 implements a speech interface that recognizes those commands and executes the appropriate actions [2,13-14]. The user can command a welding operation just by saying:

User: Robot two approach origin.
Robot: Near origin, master.
User: Robot two origin.
Robot: In origin position, master.
User: Robot two weld.
Robot: I am welding, master.
User: Robot two approach final.
Robot: Near final, master.
User: Robot two home.
Robot: In final position, master.

That's easy, isn't it?
And it makes robotic welding a fun task. Like a computer game.

4.5 References

[1] Observatory of European SMEs 2002, European Commission, 2002
[2] Pires, JN, et al, "Welding Robots, Technology, System Issues and Applications", Springer, London, 2006
[3] Pires, JN, "Semi-autonomous manufacturing systems: the role of the HMI software and of the manufacturing tracking software", Elsevier and IFAC Journal Mechatronics, to appear 2005.
[4] Microsoft Speech Application Programming Interface (API) and SDK, Version 5.1, Microsoft Corporation, http://www.microsoft.com/speech
[5] Microsoft Studio .NET 2003/2005, TechNet On-line Documentation, Microsoft Corporation, http://www.microsoft.com, 2003/2005.
[6] Bloomer J., "Power Programming with RPC", O'Reilly & Associates, Inc., 1992.
[7] RAP, Service Protocol Definition, ABB Flexible Automation, 1996 - 2004.
[8] ABB IRC5 Documentation, ABB Flexible Automation, 2005.
[9] Pires, JN, "PCROBNET2003, an ActiveX Control for ABB S4 Robots", Internal Report, Robotics and Control Laboratory, Mechanical Engineering Department, University of Coimbra, April 2004.
[10] Pires, JN, "Complete Robotic Inspection Line using PC based Control, Supervision and Parameterization Software", Elsevier and IFAC Journal Robotics and Computer Integrated Manufacturing, Vol. 21, Nº1, 2005
[11] Pires, JN, "Handling production changes on-line: example using a robotic palletizing system for the automobile glass industry", Assembly Automation Journal, MCB University Press, Volume 24, Number 3, 2004.
[12] Siemens, "S7-2000 Programmable Controller Programming Manual", Siemens Automation, Edition 08/2005, 2005.
[13] Pires, JN, Godinho, Tiago, and Ferreira, Pedro, "CAD interface for Automatic Robot Welding Programming ", Volume 31, nº1, Industrial Robot, An International Journal, MCB University Press, 2004.
[14] Pires, JN, and Loureiro, A, et al, "Welding Robots", IEEE Robotics and Automation Magazine, June, 2003

5

Industrial Manufacturing Systems

5.1 Introduction

Industrial small and medium (SME) manufacturing companies face complex and challenging market conditions that may impact their organization and economic strength. In fact, for a manufacturing SME to remain competitive in the global economy, it must cope with the following basic characteristics of the market:

- Global competition – actual companies compete on a global scale and with products from all over the world, i.e., coming from very different economic realities in terms of organization, labor, social protection and security, etc. Their competitors are global companies that address the markets with specific objectives and strategies, making the competition very unpredictable.
- Demand for more quality at lower prices – customers want the continuous improvement of quality at lower prices, i.e., customers tend to evaluate the quality of the product/service obtained for the money spent in buying it. This puts big pressure on companies since the market offers other options for the same product or service, and customers are used to making comparisons using the quality/price ratio.
- Very complex products – many of the modern high-technology products are very complex to manufacture since they often are composed of many mechanical parts, electronic components, software modules, etc. This poses new challenges to manufacturing systems.
- Very short life-cycles and time-to-market periods – competition and continuous innovation tends to reduce the life-cycle of products, forcing companies to evolve their line of products more often and with higher levels of agility.

This *scenario* poses very difficult challenges to manufacturing SMEs, namely on the quality of their manufacturing systems, in terms of flexibility and agility, and on their overall competitiveness. In fact, production plants based on human labor aren't competitive with equivalent companies located in low-salary countries. Consequently, these types of production plants tend to move their facilities to those countries or economical regions trying to take advantage of the low obligations to human labor, social security and protection, safety regulations, etc., and remain competitive in the global market. This logic has negative effects on western economies because important production sectors and jobs tend to move to low-salary countries. Consequently, the impact on the economic and social welfare is significant, working against our civilization model.

The only way to fight this trend is to focus on science and technology, developing manufacturing solutions that are flexible and agile, and that integrate efficiently with human operators. Flexibility is important to face the constant product change due to competition, fashion trends, quality requirements, and so on. But the time to market is also fundamental, which requires flexible systems that are easy to use and simple and fast to reconfigure, i.e., the modern world requires far more than flexibility and puts the focus on agility, which is a very interesting concept. Another important factor is the efficiency of the human-robot interfaces, which should allow humans and machines to operate as coworkers taking advantage of each other's abilities.

This chapter detail's a few industrial examples, with the objective of demonstrating how the concepts and ideas presented in this book can help to build manufacturing systems that are flexible, agile, and easy to use. All the systems presented were developed and built by the author of this book in cooperation with partner companies operating in Portugal.

5.2 Helping Wrapping Machines for the Paper Industry

In this section, a remote software environment developed to monitor and control robotic manufacturing cells is presented and discussed. It was used with an industrial system developed to wrap, label, and assist the storage of paper rolls coming from highly efficient paper machines. The system is also briefly introduced pointing out its main advantages. Special attention is given to the software architecture used to develop the remote services available from the system:

- Services for system monitoring
- Services for system maintenance
- Services for file and database handling
- Services for production monitoring
- Services for operator interface and system parameterization from the system control panel

The advantages of using distributed and object-oriented software approaches are also discussed, using some inside from the presented implementation. Finally, the utilization of electronic messaging services with industrial manufacturing systems is introduced and discussed.

5.2.1 Layout of the System

The system presented here was mainly designed to be used at the end of a paper machine to help with the wrapping and labeling operations of the paper rolls. Briefly, paper is produced in cylindrical rolls of several dimensions (with diameters ranging from 800mm up to 1600mm, and lengths ranging from a few centimeters to 2-3 meters) and weights. Figure 5.1 represents a diagram of the system showing its basic stations, i.e., places where robots are used to perform the required operations.

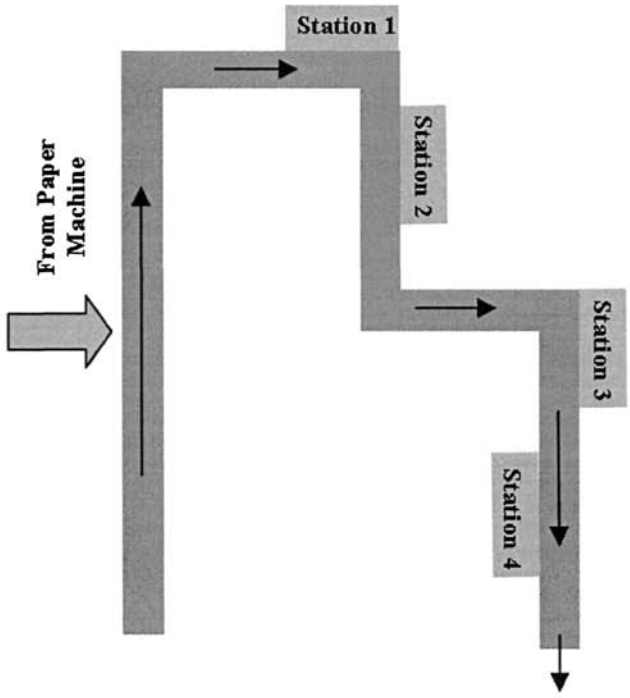


Figure 5.1 Basic organization of the of the robotic wrapping and labeling system

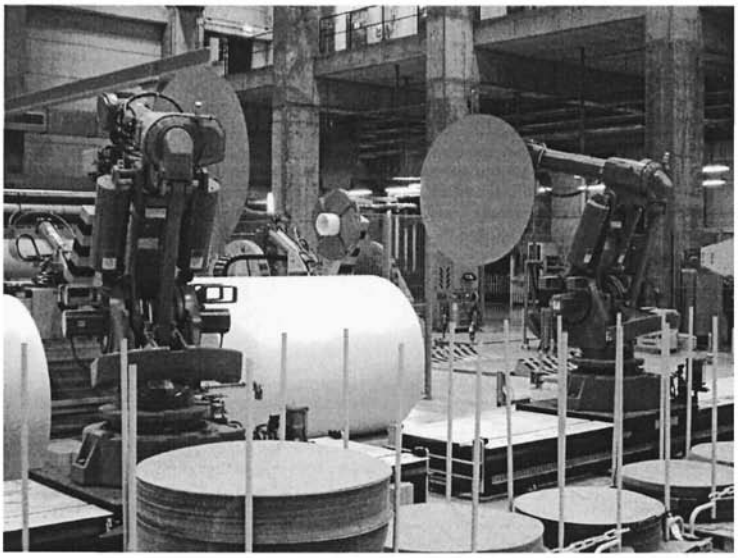
Paper rolls coming from the paper machine are labeled by a human operator using barcode sticks. The assigned code constitutes a unique identification of each roll. In the first station, the paper rolls are measured and weighted and that information is automatically inserted into the factory production database for further use, namely on the subsequent stations to pre-position the subsystems used in each station and to adapt the behavior of the local software. The system is controlled using industrial PLCs, which are accessible through *Profibus* by the PC that run's the human-machine interface software. The fieldbus network connecting the various system resources is also *Profibus*.

5.2.1.1 Station One – Dimensions and Weight

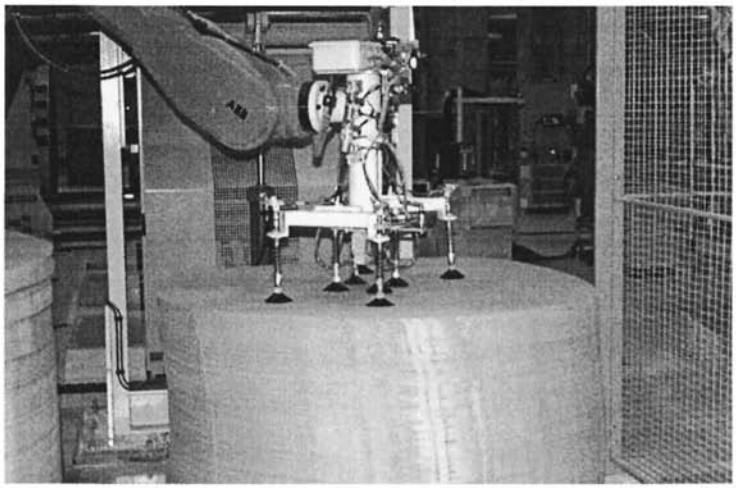
In this station, each roll is measured and weighted automatically and autonomously. The obtained values are introduced into the production database using the ID number in the barcode (barcode readers are used here). The rolls are serialized starting from this point and consequently there is no need to keep track of the rolls in the rest of the process, i.e., after this station there is no way to remove the rolls manually. The barcode numbers will be checked again at the end of the wrapping process when the rolls enter the automatic warehouse.

5.2.1.2 Station Two – Roll Wrapping and Inner Header

Rolls are wrapped using a wrapping machine assisted by two industrial robots ABB IRB6400 (equipped with the S4C+ robot controller) [1]. The robots are commanded to pick two headers, one per robot, of the appropriate dimensions (there are six piles of different headers available) and hold them against the two bases of the roll (Figure 5.2). The dimension of the header to pick is a parameter of the pick command, which is sent to each robot through *Profibus*. Consequently, a client-server software architecture is used, having the robots operating as servers. Synchronization and messaging (including error handling) with the station PLC, which also handles the wrapping machine, is done by *Profibus* using a simple IO protocol. The system is able to wrap rolls in cycles of less than 20 seconds.



a)



b)

Figure 5.2 Operation in station two: a) holding the headers, b) picking a header

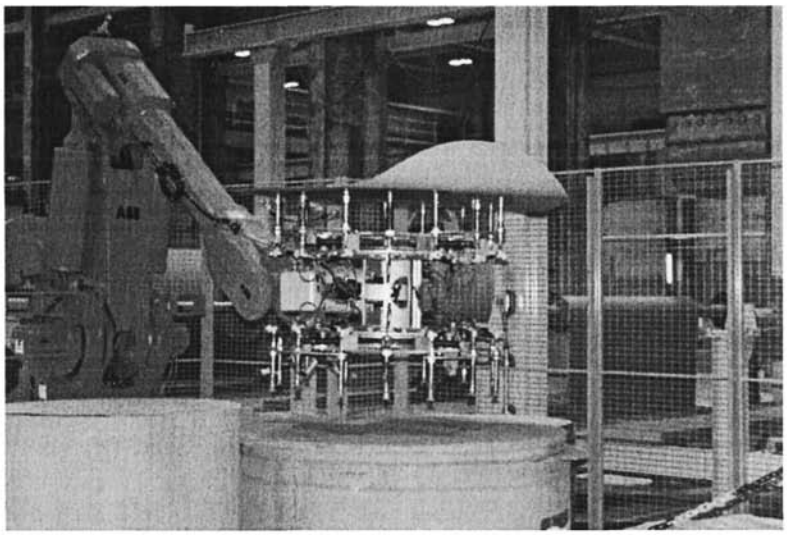
5.2.1.3 Station Three – External Header

External headers are applied on the rolls to finish the roll wrapping process and hold the wrapping paper. Operation is assisted using one industrial robot (ABB IRB6400 equipped with the S4C+ robot controller) [1]. The robot is commanded to pick two headers (gripper holds two headers) and put them, properly centered in accordance with its diameter, on the plates of a heated press. The headers are made from a type of paper that has glue impregnated in its structure. The heat makes the

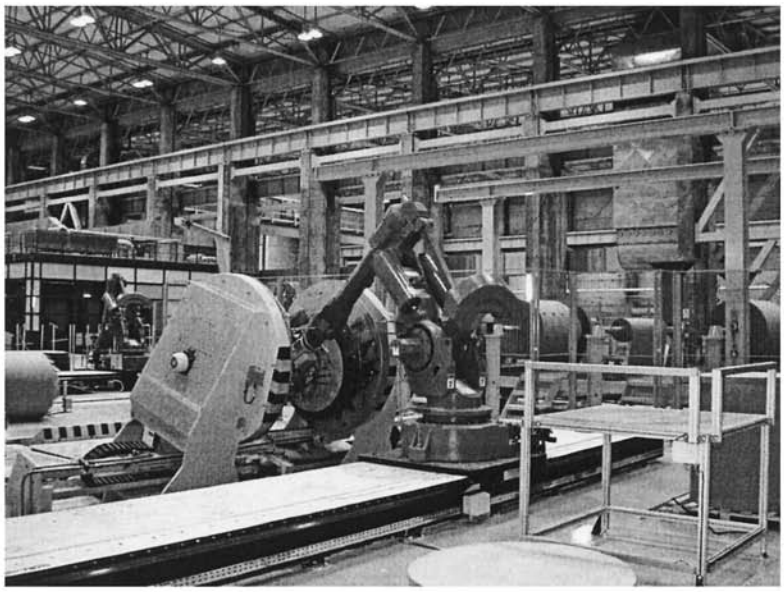
glue emerge at the surface of the header, enabling the press to glue them to the rolls just by applying pressure (Figure 5.3). Due to the cycle time requirements (less than 20 seconds per roll), the command sent to the robot to pick a pair of headers includes the diameter of the actual roll (like in the previous station) but also the actual position of the press plates (to speed up the wrapping process, the press is independently commanded to pre-position its plates as a function of roll length). Since the press is hydraulic, the position of its plates is confirmed by the robot just before entering the press workspace to place the headers. This presents robot collisions with press plates, which would eventually destroy the robot and gripper.

5.2.1.4 Station Four – Labeling

In this station (Figure 5.4), two labels are applied to the wrapped rolls (one on the top and the other on the right side of the roll) with the information about the roll printed in the label (dimensions, weight, customer, production date, etc.). Each label also has a barcode that will be used by the automatic warehouse to process the roll. Labels are printed by an office laser printer, and outputted to a small ramp. The robot picks the labels when commanded to do it, waits for the "*glue labels*" command, puts glue on the surface of the labels (using the gluing machine), waits for the roll in position, and finally places the labels on the roll. After each basic operation, the execution status is checked and the next operation is commanded only if the previous one finished successfully. If an error occurs, the current process is aborted and the error is issued back to the commanding machine (in this case a PLC).



a)



b)

Figure 5.3 Operation in station three: a) picking a pair of headers, b) placing the headers on the surface of the heated plates of the hydraulic press

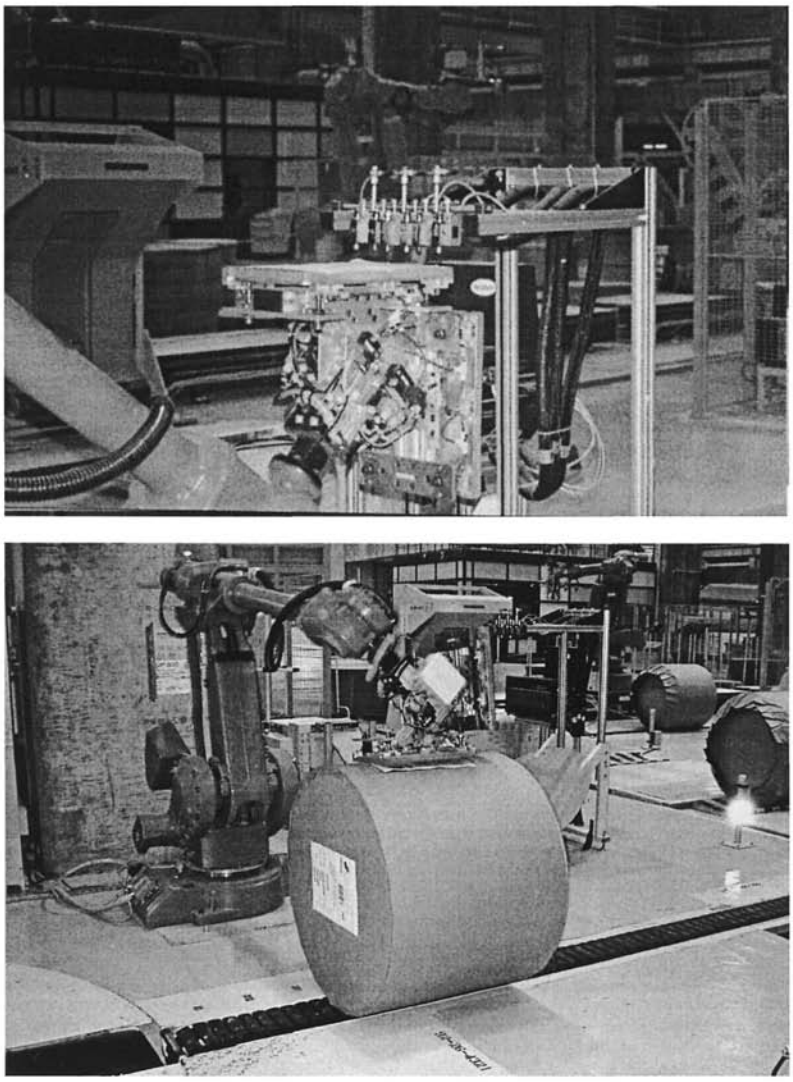
This same procedure is used in any of the other stations. All commands are acknowledged when they finish, i.e., a message specifying that the command executed correctly is sent back to the commanding machine. Communication runs over *Profibus* using a simple IO protocol.

Another version of this labeling station was built for another paper machine (see section 3.6), at the same company, that uses an *Ethernet* network and a PC to interface with the production database. The PC is also used to command the station, using *remote procedure calls* (RPC) sent to the robot controller. It is important to discuss here the basic differences between the two systems.

Considering the brief description made above and in section 3.6, and considering that robots used in industrial applications are commanded to execute very precise tasks, it is clear that in both cases there's the need for a collection of services properly designed to execute those tasks. Both systems implement a collection of services designed to execute every task available from the system. The services are implemented as generally as possible and require parameters to be properly requested by the remote client. A simple "*switch-case-do*" loop, driven by the word or number that defines the command, can be used to implement the server.

The difference resides in the way those services are requested. In the example presented in section 3.6, the services are requested using RPC calls, and in the example presented in this section the services are requested using a simple IO

protocol (see section 3.2.1). Furthermore, the version presented in section 3.6 includes an intermediate server used to connect the factory production software and the robot controller (Figure 5.5). This server listens for TCP/IP calls and simply translates the calls to robot commands, collecting the answers and sending them back to the calling machine (the production software computer).



a)

b)

Figure 5.4 Labeling system: a) tool and gluing machine, b) Robot placing label

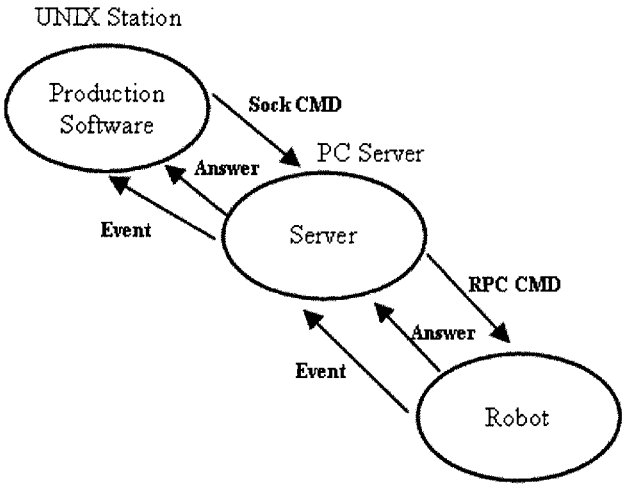


Figure 5.5 Connection between robots and factory production software: using TCP/IP sockets and SUN RPC 4.0 compatible RPCs

5.2.2 EmailWare

In every station presented, any error is logged and sent to the commanding computer as part of the answer: error codes are used to identify each type of error. Consequently, on an error situation the calling machine can decide what to do based on the received error code, for example, repeat the command.

Furthermore, every system has a checklist of basic conditions it needs to operate. For example, the labeling system needs to verify the following conditions to enter the *ready mode*:

- Air pressure at the appropriate working level
- Printing machine at the ready mode
- Glue machine at the ready mode

If a system is experiencing some type difficulty and one of the above conditions is not met, then the system enters the "*error mode*" and rejects all the incoming commands until the problem is solved.

At this point several things can be done. Let's discuss it a little bit more with an example. Suppose that there was a vacuum failure in the gripper, caused by air pressure failure (*venturi* devices are used to generate vacuum for suction cups). The system is then unable to pick and hold labels. If the problem appears during task execution, then an event may be fired (if an event firing mechanism is

available) and an error code is issued back when the command finishes (0 – success, < 0 means an error identified by the error code). The simple way to proceed and to warn operators is to act on local warning devices (a bell, a flashing light, etc), on flashing warnings on system panels, etc.

This *scenario* was the motivation to develop the *EmailWare* application, which was then extended to enable a more general task of supervising and monitoring the complete system. With those ideas in mind, a server was built to monitor an installation of robots (networked robots using TCP/IP over *Ethernet* or a serial channel) inside a factory or in a research environment. The server uses the already mentioned *ActiveX* component (*PCROBNET2003/2005*) and is capable of checking the robots available on the nework for selected interesting information, logging all events, and warning the user immediately when a selected event actually occurs. Operators are not always near the system control computer, but can be reached by beeper, mobile phone, or e-mail. In fact, they can be in an office doing some desktop job, somewhere in the plant or at home after hours. A manufacturing system should be able to reach them to send urgent information. The same situation happens with developers. They need to recollect information about their systems and sometimes, on debugging situations, they need information when certain conditions are met.

One good solution is to use short e-mail messages sent to selected accounts with brief information about events. Those accounts could be regular e-mail accounts, SMS services, beepers, etc. The application should also accept e-mail messages, coming from authorized users requesting more details about any subject (see Tables 5.1 and 5.2).

Using this application, the user may define for each robot in the installation the type of events he wants to receive. The user can also request the system to send complete reports daily, weekly, or monthly. When one of the selected events actually occurs, the application sends a short e-mail to the defined e-mail accounts. The user also selects the accounts that can receive reports, log files, or long e-mails (long e-mail should not be sent to SMS accounts or beepers).

Table 5.1 Type of events

Type of event	Parameter 1	Parameter 2	Parameter 3	Parameter 4
IO_DIGITAL	name	T0 / T1		
IO_ANALOG	name	H / L	Value	
VAR_NUM	name	H / L / C	Value	
VAR_BOOL	name	T0 / T1		
STATE_SYS	TA / TM			
STATE_PRG	TR / TS			
ERROR				
LOGS_D	type	type	Type	...
LOGS_W				...
LOGS_M	type	type	Type	...

where the symbols have the following meaning:

IO_DIGITAL – digital IO events.
IO_ANALOG – analog IO events.
VAR_NUM – events related with RAPID <num> variables.
VAR_BOOL – events related with RAPID <bool> variables.
STATE_SYS – system state events.
STATE_PRG – program state events.
ERROR – error events (any type of error).
LOGS_D – send logs daily.
LOGS_W – send logs weekly.
LOGS_M – sends logs monthly.
name – name of variable or signal (string).
T0 – transition to zero.
T1 – transition to 1.
H – Higher than value.
L – Lower than value.
C – When variable changes.
TA – transition to auto mode.
TM – transition to manual mode.
TR – transition to program running.
TS – transition to program stop.
type – type of log.

Table 5.2 Type of commands

Command	Parameter 1	Parameter 2	Parameter 3
LOGS	type	type	...
SYSTEM			
PROGRAM			
IO_DIGITAL	all / signal	signal	
IO_ANALOG	all / signal	signal	
IO_ALL			
VAR_NUM	name		
VAR_BOOL	name		
STOP_PRG	password		
START_PRG	password	AP / FB	
UNLOAD	password	name	
LOAD	password	name	
MOTOR_ON	password		
MOTOR_OFF	password		
X_CMD	password	par_1	...

where the symbols have the following meaning:
LOGS – send log files.
SYSTEM – send system state information.
PROGRAM – send program state information.
IO_DIGITAL – send information about digital IO as specified.
IO_ANALOG - send information about analog IO as specified.
IO_ALL – send information about all IO.
STOP_PRG – stops current program.
START_PRG – starts current program.
UNLOAD – unload module specified (name).

LOAD – load module specified (name).
MOTOR_ON – motors ON state.
MOTOR_OFF – motors OFF state.
X_CMD – any command implemented in RAPID.
all – all signals of this type.
password – password to execute this command (if password fails, then user is removed from list of allowed users and an e-mail to administrator is issued).

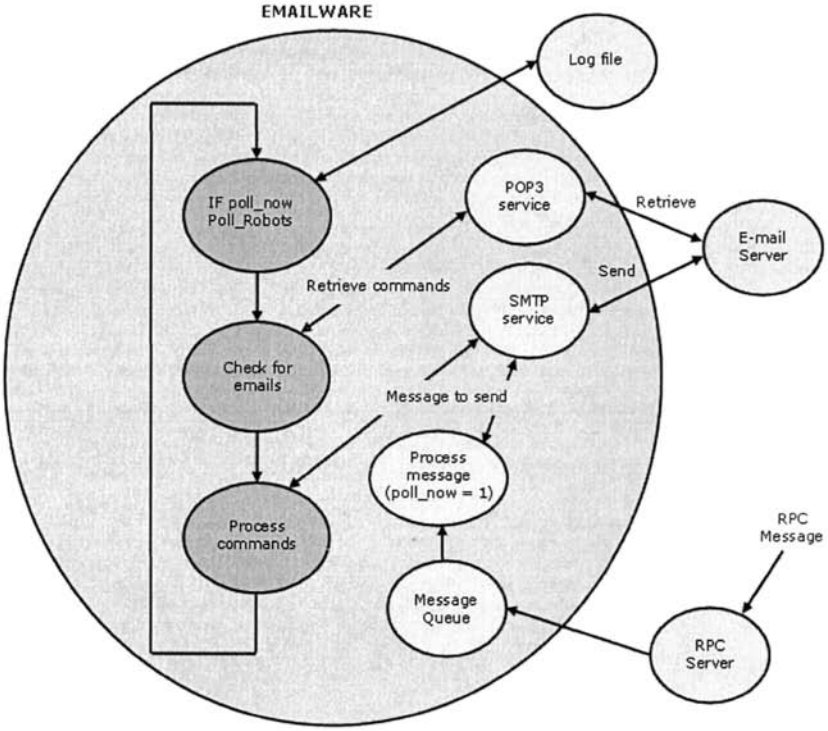


Figure 5.6 EmailWare: selecting a robot

Another important feature is the possibility to send e-mail commands to the application asking for more details on several aspects (see Table 5.2 for the types of commands that can be issued). The user can issue commands to any robot in the installation. The application checks if the sender is allowed and then processes the command. Those commands are e-mail messages sent to *emailware@company* with subject "*command*" and with the following syntax:

 # robot_dns_name command parameters

where "*robot_dns_name*" is the registered DNS name of the robot and "*command*" is a command, using the required "*parameters*" from Table 5.2. The e-mail

message can hold any number of commands (one per line starting with character '#') addressed to several robots.

The application cycle polls all robots for any change (it does not keep open clients, just opens a client connection, makes a survey, and closes the connection), fires e-mails if there is any change, and then processes commands (Figure 5.6). Since there is an RPC server working in parallel receiving asynchronous messages from any robot, any urgent event is immediately attended and information is issued to the user (the information is sent once when it happens, i.e., when the event is fired from the robot, and a second time when the polling process detects the change). The polling frequency of the robots can be adjusted to avoid overloading the system, ranging from 1/10 Hz (higher frequency) to 1/60 Hz (lowest frequency).

5.2.2.1 EmailWare Application Example

To show the potential of this tool, lets give a simple example. Suppose that at some industrial installation there is a robot (named "*babylon5*") doing arc-welding operations. Suppose also that the welding software keeps information on the number of pieces that have been welded (*num_pieces*), on the amount of time in operation (*opr_time*), and on the idle time (*idle_time*). There is also information on how many errors were encountered during operation (*num_error*); it is considered here that the system can handle and maybe automatically recover from certain operational errors (consequently, for each error the *num_error* variable is incremented and an operational message is issued like: *bad* or *no piece in place*, *no gas*, *no air pressure*, etc), which is normally the case. There are also some IO inputs and outputs like: gas information (digital input, *gas_on*), air pressure information (digital input, *air_on*), wire information (digital input, *wire_on*), etc. Finally, suppose that the user wants to have daily reports about the system, including the state of some of variables.

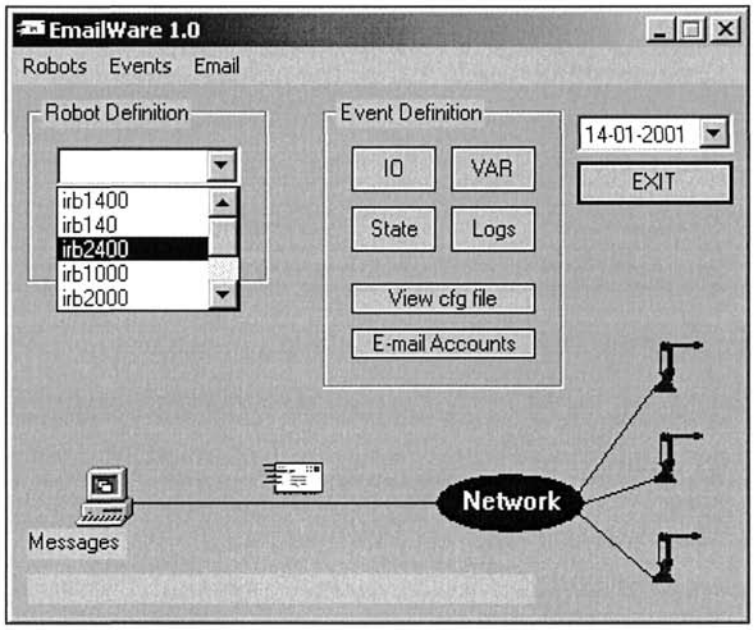


Figure 5.7 EmailWare: selecting a robot

To configure *EmailWare* for the welding application, the user starts by selecting the robot from the available robots (Figure 5.7). After that, the user selects the IO signals, the variables, and the type of system states of interest.

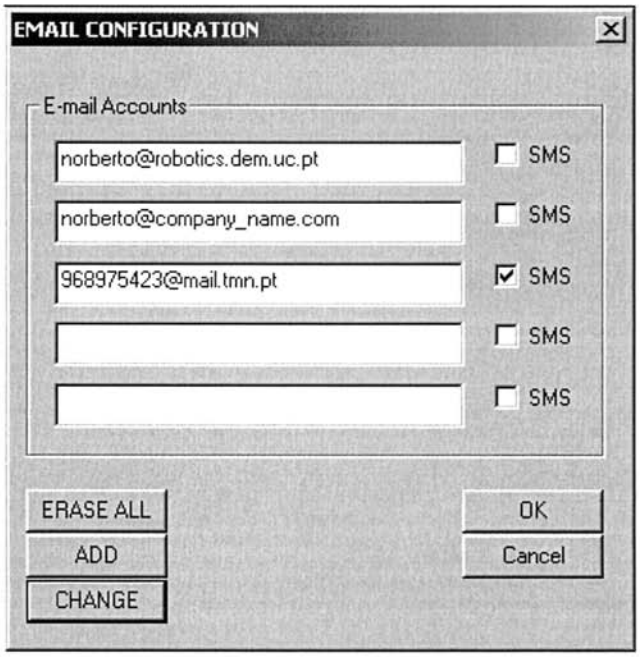


Figure 5.8 EmailWare: dialog to define e-mail accounts

Then the user e-mail accounts (Figure 5.8) must be defined (up to five accounts) and the ones that can receive long e-mails (the user should identify at least one normal e-mail account and one SMS account) must be specified. All the configurations are stored in a configuration file (*rob_conf.cfg*) that can be accessed using any text editor (*Notepad*, *Wordpad*, *Word*, etc). For the above-mentioned example, the file could look like the one in Figure 5.9.

As mentioned already, the application was tested on the industrial installation, presented in this section which uses four robots, but the interested reader can make his own test using our laboratory robots. Just visit the *EmailWare* web site located at *http://robotics.dem.uc.pt/emailware/* and sign up to receive warnings about the operation of one of our robots. Interested readers can also send commands to it. The site is a demonstration site, so only a few features are demonstrated and users cannot customize them. Finally, a demo version that is fully operational for one robot only (robot serial number is needed) may also be requested.

```
* EmailWare Header
* (C) J. Norberto Pires 2000-2006
* norberto@robotics.dem.uc.pt

* USER DEFINITION
norberto@robotics.dem.uc.pt
norberto@company_name.com
968975423@mail.tmn.pt SMS
* ROBOT DEFINITION
name = babylon5
domain = dem.uc.pt
IP = 193.136.213.69
Model = ABB_IRB_1400
@
IO_DIGITAL 3
gas_on T0
wire_on T0
air_on T0
IO_ANALOG 0
VAR_NUM 3
error C
opr_time H 100
idle_time H 50
STATE_SYS TM
STATE_SYS TA
STATE_PRG TS
STATE_PRG TR
LOGS_D all
&
* ROBOT DEFINITION
name = perseus
domain = dem.uc.pt
IP = 193.136.213.61
Model = ABB_IRB_2400
@
...
&
*End of configuration file
```

Figure 5.9 Example of configuration file (*rob_conf.cfg*)

Consequently, any of the specified users receive messages (by e-mail or SMS) about the programmed events that can look like:

Babylon 5: Ei guys, I'm stopped, no air-pressure or air-pressure too low.
Babylon 5: Ei guys, I'm stopped, no wire.
Babylon 5: Ei guys, wire is running out.
Babylon 5: OK, air-pressure is on again.

5.2.3 Conclusions and Discussion

The system presented in this section is commanded remotely from the PLC used to manage the operation of the cell. The system also uses a PC to interface with the operator, and updates and retrieves information from the factory production software. The system was designed to operate almost autonomously, i.e., with minor operator intervention limited to error and maintenance situations. Consequently, a client-server software architecture was used, with the robots working as servers allowing remote clients to explore and operate the system. This proved to be a nice solution capable of providing a good performance and high levels of flexibility, because the system's basic operation is defined by the operating software. Adding new functions or changing the operation is an easy task and in fact was done several times to adjust to new requirements.

Finally, a simple e-manufacturing solution was introduced in this section. It enables operators to receive operation events when they occur, allowing a more efficient supervision of the system, reducing down time due to errors or unavailability of certain operating conditions. This idea of having automation equipment sending messages to users with relevant information about its current status, and enabling users to request more details and sending a few commands, also by e-mail, can be extended to other areas: monitoring warehouse systems that could inform users about critical points, smart houses informing users about current situations and enabling some remote commands, remote maintenance, and so on.

5.3 Complete Robotic Inspection Line for the Ceramic Industry

Non-flat ceramic products, like toilets and bidets, are fully inspected at the end of the production process to search for structural, surface, and functional defects. Ceramic pieces are transported to the inspection lines assembled in pallets, carried by electro-mechanical fork-lifters or *automatic guided vehicles* (AGV). Pallets need to be disassembled, feeding the inspection lines where human operators execute the inspection tasks. Also, the pieces that pass inspection need to be palletized again in the final pallets used for product distribution. Those de-palletizing and palletizing operations are physically demanding so they are good candidates for robots.

This section is a case study on the development of a collection of prototype manufacturing cells, designed to perform automatic palletizing and de-palletizing operations of non-flat ceramic pieces such as toilets and bidets. The factories of these types of products show an impressive mixture of human and automatic labor, meaning that special attention must be taken with regard to human machine interfaces (HMI), safety, mode of operation, etc.

Non-flat ceramic products are commonly used in our homes and are mainly associated with personal care tasks. The industrial production of these ceramic products poses several problems to industrial automation, especially if robots are to be used. Basically, these problems arise from the characteristics of the ceramic pieces: non-flat objects with high reflective surfaces, very difficult to grasp and handle due to the external configuration, heavy and fragile, extensive surface sensitive to damage, high demand for quality on surface smoothness, etc. Also, the production setups for these types of products require high quality and low cycle times, since this is a large scale industry that will remain competitive only if production rates are kept high. Another restriction is that this industry changes products frequently, due to fashion tendencies in home decoration, etc. Also, there is the mixture of automatic and human labor production, which is a difficult problem since HMI are very demanding and a key issue in modern industrial automation systems.

It was proposed by the partner company to build several de-palletizing and palletizing solutions, with a simple graphic operator interface, to install in their final inspection lines. In those lines human operators inspect all pieces by hand to find functional and surface defects (computer vision solutions for inspection). The challenge was to build highly efficient systems, capable of handling more pieces a day than its human counterparts, that could be easy to set up and start up at the beginning of the day. So, there is a robotic challenge and a software challenge, namely, in designing human-machine interfaces for operators.

The system presented here (Figure 5.10) was designed to take advantage of computers and available tools to parameterize and monitor an industrial robotic cell, i.e., to make human-machine interface. In the process of describing and discussing the system a few available, a few technical details are highlighted. This is also important due to the fact that all the software was built from the scratch [2], without using any of the available commercial software packages (Section 3.2).

5.3.1 Motivation and Goals

The problem addressed in this example is the construction of a complete system to assist humans in the task of inspecting non-flat ceramic pieces. Those pieces (bidets and toilets, mainly) reach the inspecting site directly from the high temperature oven, organized in pallets (input-pallets), using fork-lifters. A few operators placed along two inspecting lines (15 meters long each), inspect all the pieces by hand, searching for pieces with functional and surface defects, removing from the inspection lines the pieces rejected [3, 4]. Consequently, in this system there is the need to de-palletize the input-pallets, feeding continuously the two inspection lines. The system must also pick the accepted pieces from the end of the inspection line, palletizing them again into the pallets (output-pallets) used for product distribution (Figure 5.10).

The system should work also as autonomously as possible, requiring only minor parameterization at the beginning of the work day or production cycle. The system should be able to work with input-pallets composed of four levels of ceramic pieces, eight pieces per level placed in a special order to keep pallet equilibrium, and with levels separated with pieces of hard paper. It should also be able to work with output-pallets up to five levels of ceramic pieces, eight pieces per level placed in the same order as in the input-pallets, with levels also separated by hard paper. The rule used to arrange the pieces in the pallet is to place them alternatively one up – one down, starting from the ground level, then swap to one down – one up in the next level (Figure 5.11), and keep the procedure in the proceeding levels.

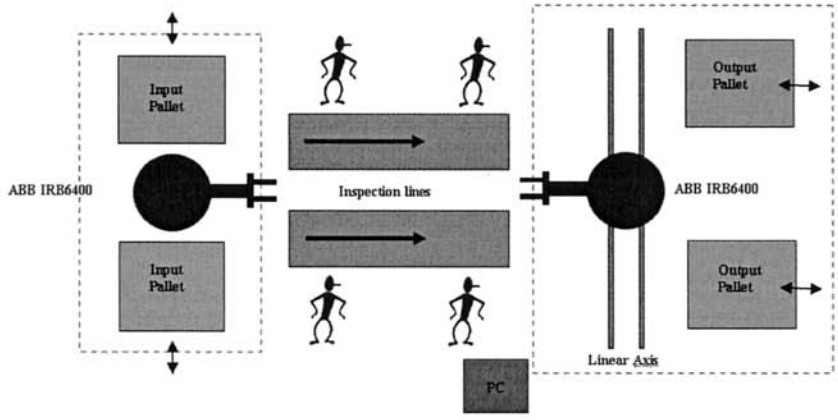
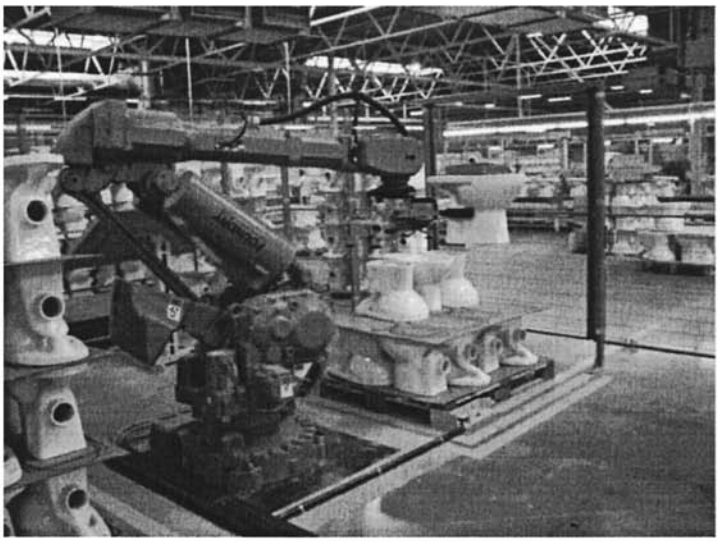


Figure 5.10 Components of the system

Actually, input-pallets are assembled manually by operators at the end of the high temperature oven. This means that the robotic system must be tolerant with possible medium-large palletizing errors, coming from misplaced pieces both in position and orientation, and also showing significant variations from level to level. Another important factor is that pallets are fed into the system by human operators using electro-mechanic fork-lifters, which also introduces some variation in the pallets. Sometime in the future, AGVs will be use to fulfill the task, reducing considerably the variations introduced and increasing the efficiency of the system.

a)



b)

c)

Figure 5.11 Pallets and view of the system: a) input pallets and de-palletizing robot; b) aspect of the de-palletizing gripper; c) view of the complete system

The main objectives for this system are summarized as follows:

- Build a complete robotic system capable of performing de-palletizing and palletizing operations to assist inspection lines
- The system must perform each of these operations in less than 12 seconds per piece
- The system should cope with high palletizing errors on the input-pallets, since they are assembled by human operators which permits to anticipate small-medium placement errors (up to 5cm in position and up to 5º around the vertical axis)
- The system should cope with deviations on the dimensions of the pieces of up to ±1 cm in each direction. Ceramic pieces grow inside the high temperature oven, making these deviations expected due to temperature deficiencies, variation of time inside the oven, variations in the ceramic mixture, etc. These deviations are not necessarily errors, but instead a characteristic of this type of production
- The system must work with pallets, both input and output, with variable numbers of pieces, ranging from any number of pieces, in the case of the input pallets, to 8, 16, 24 or 40 pieces, in the case of the output pallets

- The system should maintain information about its surroundings, so as to warn about inconsistencies between what is ordered and what is available
- The system must be parameterized easily, using a graphical interface implemented with a touch-screen. A few commercial software packages are available in the market. Nevertheless, our option was to build our own solution since the human-machine interface plays a crucial role in the performance of the system, including operator acceptance. It is therefore very important to have full control over the developed software
- The system must be optimized for each model of ceramic pieces. This means that there should be the option of introducing new models using a teach strategy

Considering these above mentioned objectives, the following challenges were identified:

- To build a human-machine interface, easy to use and capable of handling production needs. System warnings and errors must be issued to the operator's attention in an efficient way. All operations and messages must be logged for future analysis;
- To build a system capable of meeting the planned requirements;
- To explore the capabilities of the current personal computers, operating systems, and related tools on a very demanding industrial environment;

Taking the above objectives and challenges, and considering the fact that this is an industrial project, meaning it is supposed to work 24 hours a day without problems, it was decided to distribute the software to all the components of the system. A client-server architecture [2-8], based on remote procedure calls (RPC) [9], was adopted, with the PC as the client of the rest of the components of the system, including the robot controllers, and also as the interface to the operator.

5.3.2 Approach and Results

The objectives and requirements of this project necessitated a robotic cell that could handle the ceramic pieces under consideration. Proper grippers and layouts were designed and built. It was also necessary to operate the system through an external personal computer, using the teach pendant of the robot only for a few special routines not performed in every day normal operations. The robots work as slaves to that central PC, where all the parameterization is performed. The PC also monitors the operation, being of guidance when something wrong happens. The operator is able to solve problems from the PC. There is one PC for each robot, which was done for practical reasons, but it is not a requirement.

A client-server software architecture was adopted. The robot controller software works as a server, exposing to the client a collection of services that constitute its basic functionality. A collection of services was designed to fulfill all the tasks required of the system, so that they could be called from the PC (Figure 5.12). The

software architecture used in this work was presented in detail elsewhere [2 -8] (see also Section 3.2), and is distributed using a client-server model based on software components (*ActiveX controls*) [10-11] developed to handle equipment functionality.

The system is completely operated using a graphical panel running on the PC, built using the above mentioned *ActiveX controls* in *Visual C++ .NET 2003* [12]. When the system is started, the operator needs only to specify what product model will be used in each pallet, and if first pallets are fully assembled. This need is only for the de-palletizing subsystem, because there is no identification on the pieces (they are coming from the high temperature oven). On the palletizing subsystem, there is no need to specify the model, because the pieces carry barcodes, inserted by the inspecting operator, that are used by the subsystem with the help of barcode readers.

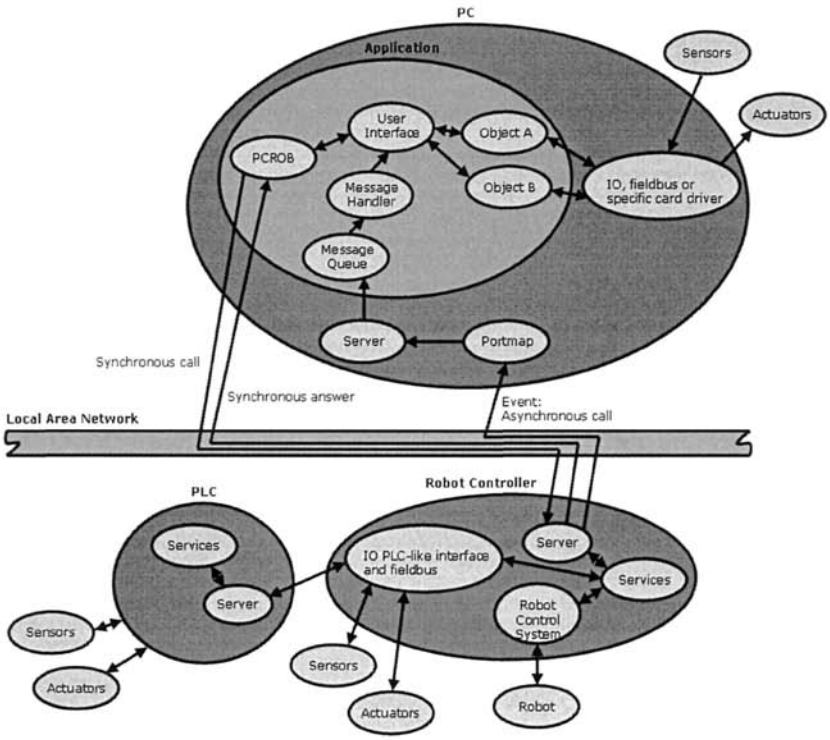


Figure 5.12 Software architecture used in this example

Sometimes, there are some non-fully assembled input-pallets on the shop floor that need to be introduced into the system. To do that, the software allows the operator to specify the position and level of the first piece. That is, however, only possible on the first pallet, because the system resets definitions to the next pallets to avoid accidents, i.e., proceeding pallets are assumed to be fully assembled. The same

happens with output-pallets, since the system must be able to fill a pallet not completely filled on the last production cycle for that model.

5.3.2.1 Basic Functioning of the De-palletizing System

When the operator commands "*automatic mode*" the robot approaches the selected input-pallet in the direction of the actual piece, searchers the piece border using optical sensors placed on the gripper, and fetches the ceramic piece. After that, the robot places the piece in the first available inspection line, alternating inspection lines if they are both available, i.e., the robot tries to alternate between them, but if the selected one is not available then the other is used if available. If both inspection lines are occupied, the robot waits for the first to become available.

Figure 5.13 shows the interface used by the operator to command the system and monitor production. It shows the commands available, and the online production data that enables operators to follow production. All commands and events are logged into a log file, so that production managers can use it for production monitoring, planning, debugging, etc. The system also uses a database, organized in function of the model number, where all the data related to each model is stored. That data includes type of piece, dimensions, height where the gripper should grab the piece, average position of the first piece of the pallet, height of the pallet, and so on. Accessing and updating the database is done in "*manual mode*", selected in the PC interface.

There is a "*teaching*" option that enables operators to introduce new models and parameterize the database for that model, where a "*teach by showing*" strategy is used. When that option is commanded, the robot pre-positions near the input-pallet and the operator can jog the robot using function keys to the desired position/orientation. Basically the de-palletizing operation is preformed step-by-step and the necessary parameters acquired in the process, asking the operator to correct and acknowledge when necessary. The operator is asked to enter only the "*model number*" to teach, the height, and the width of the piece. The rest is automatic. After finishing this routine the model is introduced into the database, and the system can then work with that model number.

The system is able to check for errors such as: wrong pallet for model, presence of pallet, model not known, no piece in place, wrong level, etc. Proper warnings are sent to the PC for operator information, and displayed using software icons and short messages.

5.3.2.2 Basic Functioning of the Palletizing System

A similar approach was used for the palletizing operation. Two inspection lines are also used, with the robot trying to alternate between them. But the first available piece is removed not slowing down production. A similar approach to the one used in the de-palletizing sub-system is used to "*teach*" models to the robot. Also, the system identifies the model number from the piece barcode when "*automatic*

mode" is commanded, fetches the piece, and inserts it in the pallet compatible for that model. The operator is able to select what pallet to use first, how many pieces are already there, and how many pieces it should carry (Figure 5.14). Do to the required dimensions of the output-pallets, the robot was placed on the top of a linear axis, controlled by the robot control system (robot external axis), so that a wider area could be reached. The system is also able to check for errors such as: wrong pallet for model, presence of pallet, model not known, no piece in place, etc. Proper warnings are sent to the PC for operator information, and displayed using software icons and short messages.

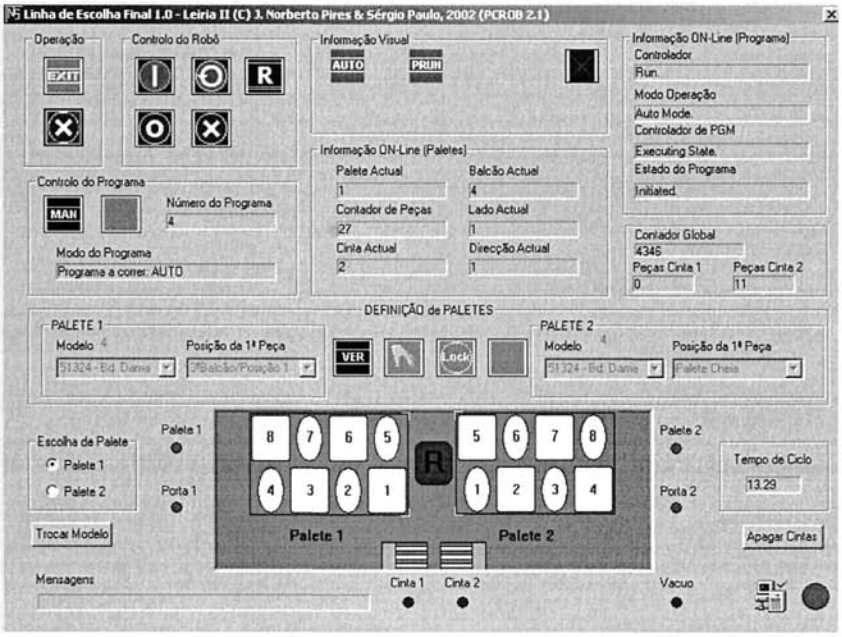


Figure 5.13 Example of an interface used by operators (de-palletizing system)

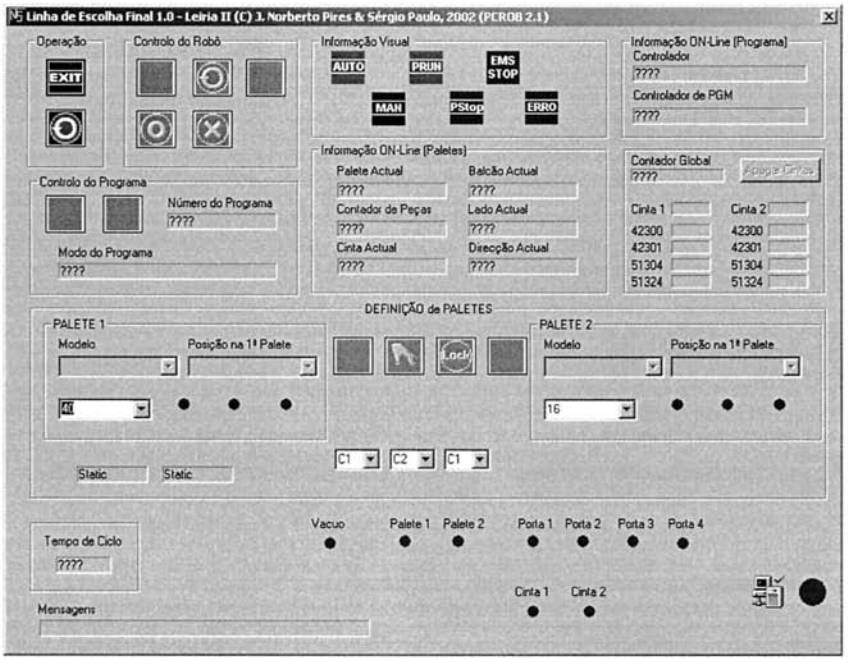


Figure 5.14 Example of an interface used by operators (palletizing system)

5.3.3 Operational Results and Discussion

The system achieved the required operational results and is flexible enough when introducing new models. Currently it works two shifts a day, almost autonomously, making around 1400 pieces per shift (one shift is seven and-a-half hours). Operators adapted easily to the system, and found the touch-screen interface easy to use.

The company improved production quality and reduced production costs: fewer operators are needed and production is more efficient (more pieces are handled a day). This can be demonstrated by operational results, and also by the fact that new systems followed this one to handle other type of pieces and other types of operations, creating a strong connection between our university and this company.

A few innovations and technology transfers were successfully introduced with this project and others are ongoing with the same company [2-5]. An interesting human-machine interface for robotic manufacturing cells was introduced with good results [2-5]. The solution has been developed from the scratch using *Visual C++ .NET 2003*, constituting a software platform that can be used with other applications. Experience with operators is positive, showing that they adapted well and really enjoy using it. Nevertheless, new developments are necessary so as to

guide operators and reduce operator training. This means that advanced help should be available to guide the operator when inconsistencies are detected. Such inconsistencies include, for example:

- Commanding "*automatic mode*" without reviewing the pallets parameterization. That could be correct in some situations and consequently, allowed. At the moment only a visible warning is issued, but in the future only some sequence of operations will allow "*automatic mode*"
- Ordering a "*RSTART*", i.e., proceed with current configuration and from the same program position, after a system stop due to an error or operator manual stop. Actually this situation is permitted, after confirming the password, because we still rely on operator training and judgment. Nevertheless, in the future, operators should be guided to follow a certain procedure, reviewing actual status, so as to avoid mistakes. This can certainly be done, for example, using an inference mechanism based on *fuzzy logic*

The two presented situations are good examples of needed future developments. For a certain industrial robotic cell characterized by a set of available operations, a collection of routes should be defined considering all possible operational situations. Consequently, an operator can command the robotic cell if he follows one of those routes. This will increase safety, avoid errors, and improve efficiency. At the moment, critical operations require operator confirmation with a password, and visible warnings issued to the screen.

Another interesting innovation was the utilization of a client-server architecture, explained elsewhere [2-5] (see Section 3.2), developed by the first author, to be used with robotic cells. Using this architecture implies the clear intention to distribute functions to all "*intelligent*" components of the robotic cell, leaving to the central PC (the client) the tasks of making the service request calls, properly parameterized, and displaying system information to the user. The PC is the user's commanding interface, and his window to the system.

5.4 Handling Production Changes Online

In this section, the problem of handling production variations online, i.e., during actual production, is addressed. These variations may occur when it isn't possible to exactly guarantee working conditions during a production cycle or between two consecutive cycles. These variations are common in some types of industries, like the glass and ceramic industry, where the products may change slightly during the production cycle. Also, these industries are multi-model industries in which the production equipment is required to handle several different models of products that have their own production requirements. Since it is common to have two or

more different model campaigns during a working day, it should be possible to easily parameterize the production system when a new campaign is started.

Consequently, this section uses a highly efficient robotic palletizing system, developed for a partner company, to introduce and explain how these problems may be addressed. It includes details about practical implementation, along with a discussion of options and obtained operational results, which show the system to be a good example of human-machine cooperation.

As is common in several industries, the intermediate products need to be palletized in several stages of the production cycle, to circulate between manufacturing cells, be sold to other companies (white-line or undifferentiated products) that finish the production cycle adding their own characteristics, or to be stored inside the company in accordance with the defined production planning and company needs.

This case, the products are several models of automobile side-window glass. The palletizing system is placed after the glass cutting and washing cells. The obtained pallets are to be used in the manufacturing line that introduces the characteristic curvature of the glass. This line, which includes a high-temperature oven and an incurving system, is shared by all models of side-window glass produced by the company, which makes the task of automatically feeding the line from all cutting and washing lines very difficult to manage. Consequently, the glass is palletized using a robot manipulator and de-palletized near the incurving manufacturing line by another robot. This enables the company to handle all types of models in a very simple and efficient way.

5.4.1 Robotic Palletizing System

The system used in this example was developed to pick side-window glass from the production line and palletize it into pre-configured pallets. The system, depicted in Figures 5.15 and 5.16, is made of the following components:

- An industrial robot ABB IRB 4400, equipped with the 2002 version of the ABB S4C+ robot controller
- A PLC Siemens S7-300, to control all the systems peripheral to the robot.
- A centering system, placed on the production line, that guarantees that glasses are centered and placed in a known position before being picked by the robot
- A pneumatic gripper with retractile contact sensors and suction cups, capable of picking glasses and measuring the pallet characteristics
- A rotating system that supports two pallets, ensuring that a new empty pallet is immediately fed into the system when the previous one is full
- A computer for supervision and control, and for implementing also the human-machine interface

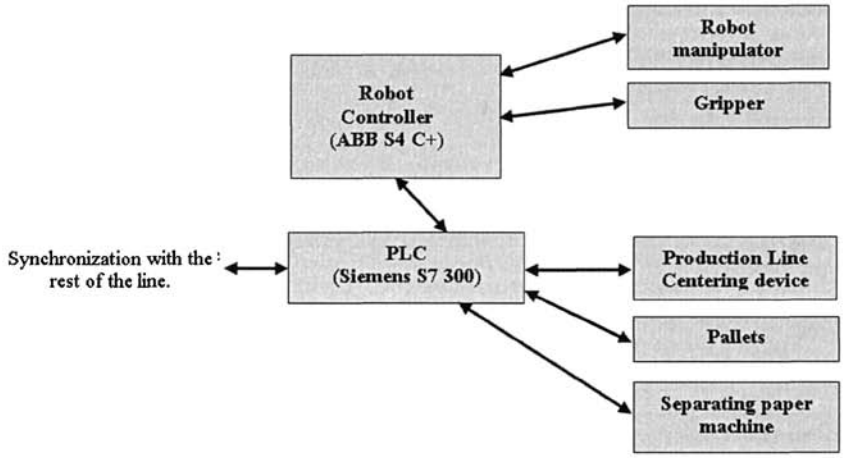


Figure 5.15 Components of the palletizing system for the automobile industry

The cycle executed by the system (Figure 5.17) is composed of the following principal tasks:

5.4.1.1 Identify Empty Pallets and Measure Parameters of an Empty Pallet
An empty pallet needs verification to measure the following pallet parameters: angle of the back of the pallet with the vertical axis, angle of the base of the pallet with the horizontal axis, height of the base of the pallet relative to the robot world reference system, and the pallet dimension. These four values change significantly from pallet to pallet and need to be obtained each time an empty pallet is introduced in the system. This task is fundamental for the success of the palletizing task, because it enables the system to place the glass always in the same conditions: at the same height relative to the pallet base and at the same distance from the previous glass. This avoids adding defects to the glass, namely small scratches on the surface of the glass (due to slipping between consecutive glasses), or on the edges that contact with the surface of the pallets (due to releasing the glass more than 1-2mm high from the surface of the pallet).

Any empty pallet needs to be measured for the above mentioned parameters that will be used during the palletizing process using that pallet. Every time the rotating base introduces a new pallet, optical sensors, placed behind the back of the pallet, detect if the pallet is empty and trigger the measuring process.

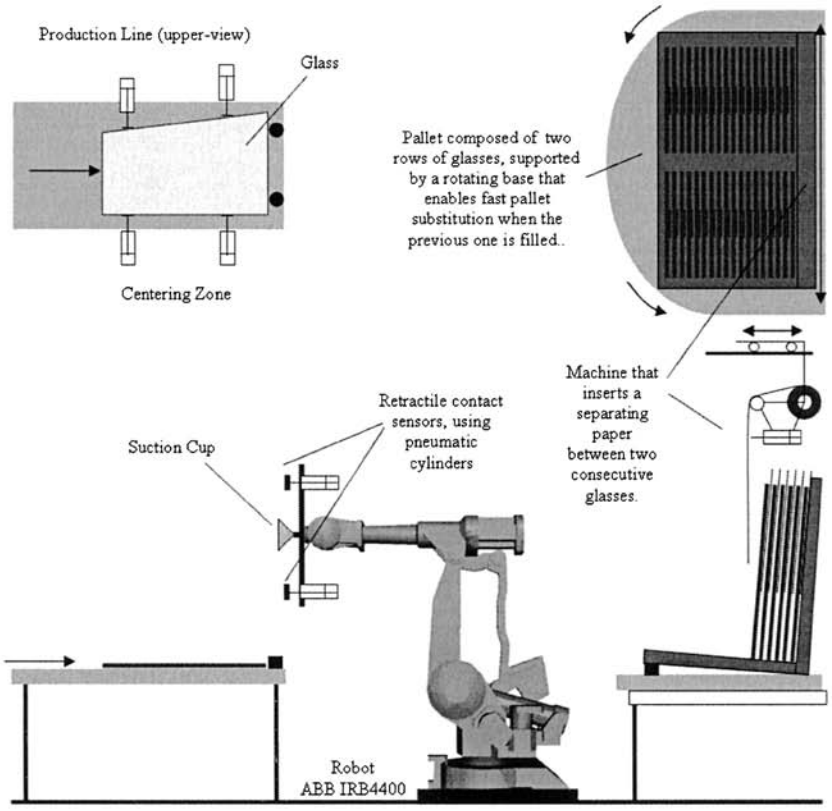


Figure 5.16 General view of the palletizing cell

5.4.1.2 Pick a Glass from the Production Line

After getting information from the PLC that there is a glass available in the production line, properly centered and in position, the robot is commanded to pick the glass from the predefined picking position (based on the glass model) and take it to a position near the entrance of the pallet.

5.4.1.3 Palletize the Glass

The glass must be placed in the row in use, taking into consideration the number of glasses already palletized and the pallet parameters. This operation means also knowing the thickness of the glass in a way to maintain the same palletizing conditions for all glasses. At the end, when a pallet is full, the robot signals the PLC that the pallet is full and places itself in a non-collision situation with the pallet, enabling the PLC to start the rotating motion that will exchange the pallets (Figure 5.18).

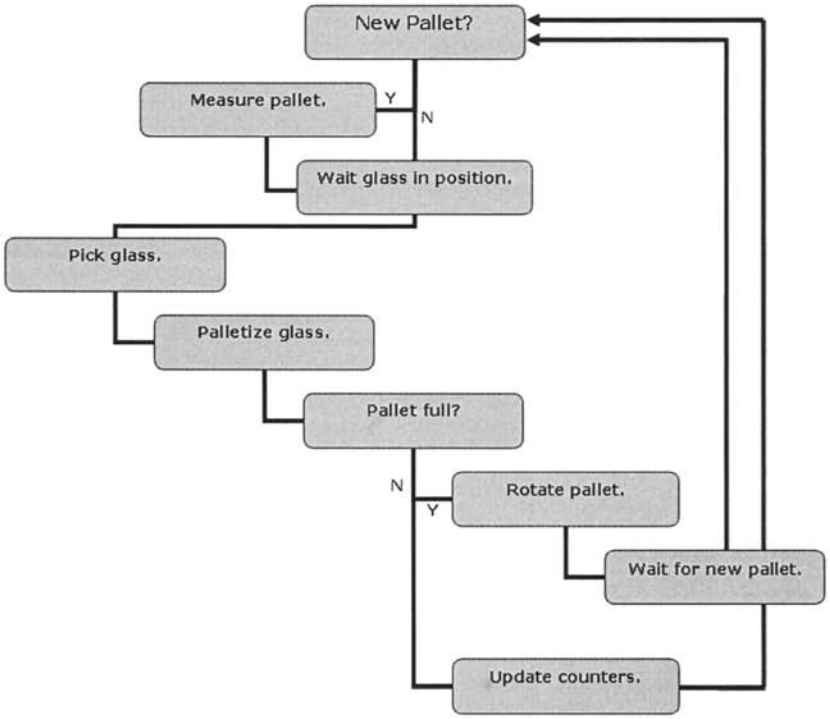
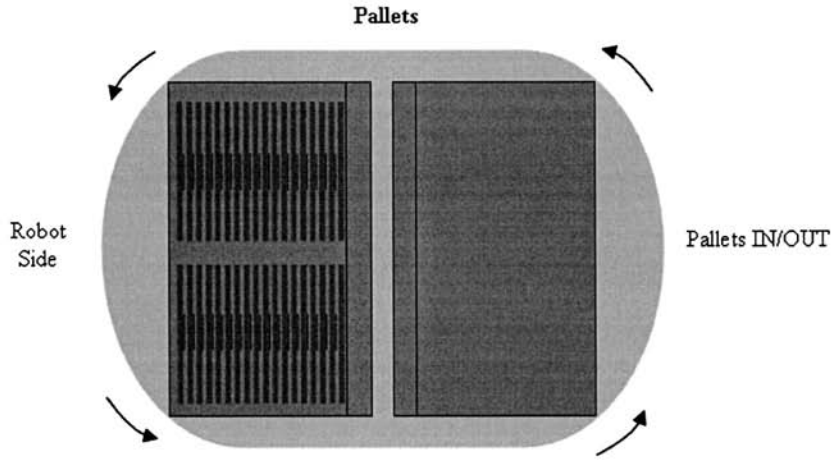


Figure 5.17 Palletizing cycle executed by the robot in automatic mode



5.4.2 System Software

Considering that the above presented system was developed to work with several models of glass (up to 128 different models), that require their own configuration in the tasks of picking and palletizing each glass, i.e., these tasks are model dependent, the operating software should explore the teach-pendant capabilities in the phase of teaching a new glass model to the system. Consequently, the software was designed to have two operating modes: manual and automatic.

Manual Mode – In this mode, all subsystem testing and maintenance routines are allowed (Figure 5.19). The user is also allowed to teach a new model to the system. This means that the robot will follow pre-determined motions, asking the operator to adjust positions using function keys. In the process, the software acquires the necessary data to completely handle that model of glass. In this mode, the production line is not operational, because production is deactivated. The robot is commanded from the robot teach-pendant (or console), using local software designed to assist the selected functions. For practical reasons, this "*manual mode*" software will not be explained further here.

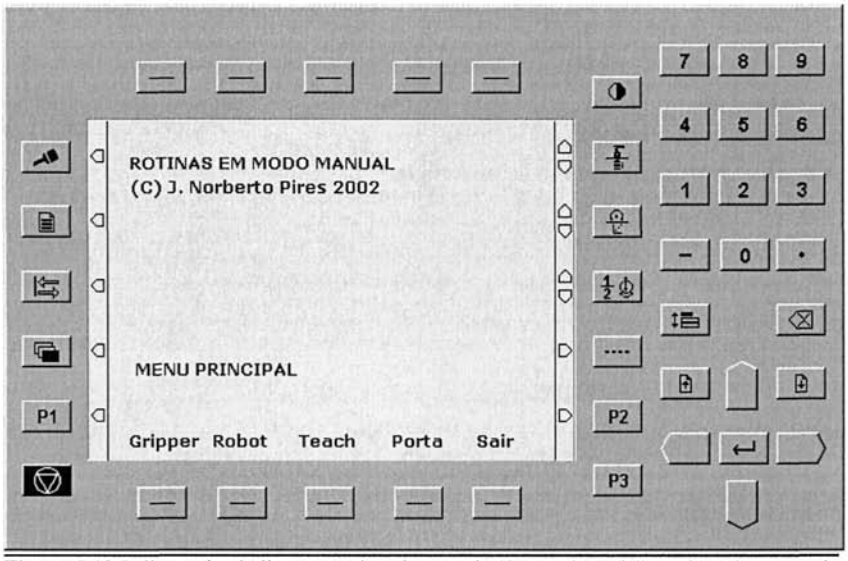


Figure 5.19 Pallet main shell presented to the user in "Manual Mode" on the robot console (original software with Portuguese interface)

Automatic Mode – The production line is placed in automatic mode and the robot should follow the cycle presented briefly in Figure 5.17. The robot uses the definitions stored in the database to handle the model selected by the operator, using the parameterizations he chooses.

The software developed to interface with the operator runs on a remote computer, connected to the robot controller by *Ethernet*. The software was developed in *Visual C++ .NET 2003* [12], using an ActiveX control [10-11] designed by the author to work with industrial robots [2-5] (see Section 3.2). The shell presented in Figure 5.20 is the operator interface to the system.

To initiate the system, the user must run the robot program using the operator interface. A "*start_program*" remote procedure call (RPC) [9] is issued, launching a computer program that implements a collection of services that can be requested from the PC using RPCs. After being initiated, the robot program waits for the selection of the operating mode, i.e., waits the user to command "*Automatic Mode*", where the robot is controlled by the system PLC using the parameterization selected by the user, or "*Manual Mode*" where the robot is commanded from the robot teach-pendant. Both operating modes may be considered as services that the robot (*server*) offers to the PC/operator (*client*). During the "*mode selection state*", where the robot waits for the user to select the operating mode, it is possible to access the system database where the definitions for each model are stored. Access to database is not allowed in any other situation, for safety reasons. Consequently, before selecting the operating mode, the user should select the model he wants to produce and parameterize the production: thickness of the model, number of pieces per row and per pallet, and the dimension of the glass. The thickness and dimension of the glass are characteristics of the model registered in the database, and consequently are not to be changed by the user. A password is required to change them.

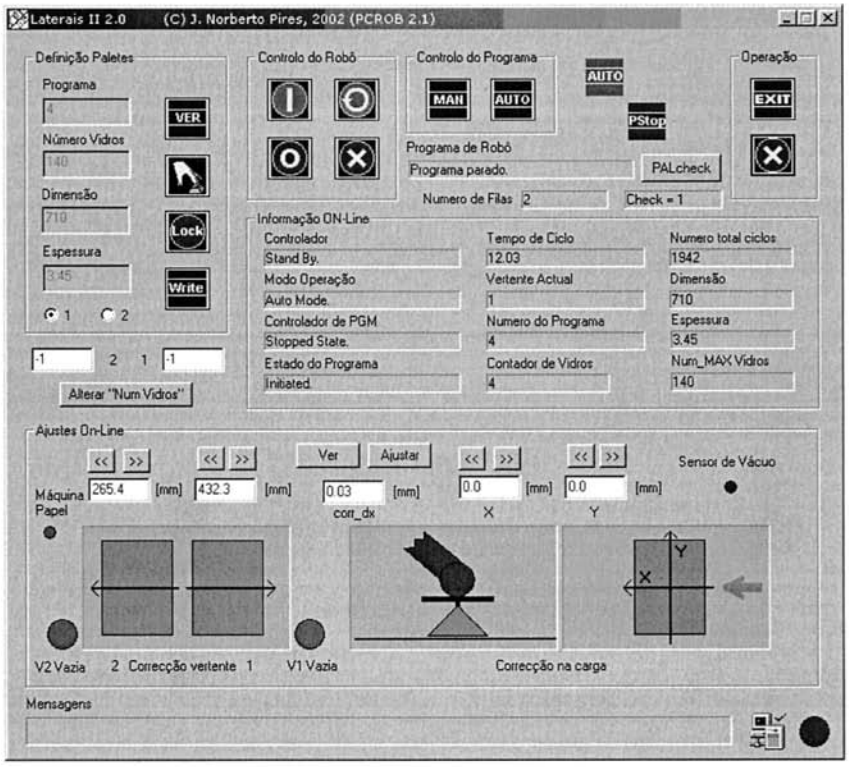


Figure 5.20 – Operator interface running on the PC (original software with Portuguese interface)

Using the interface presented in Figure 5.20, the operator is allowed to command three types of operations: Access the glass model definition database, control the robot program, and online monitoring.

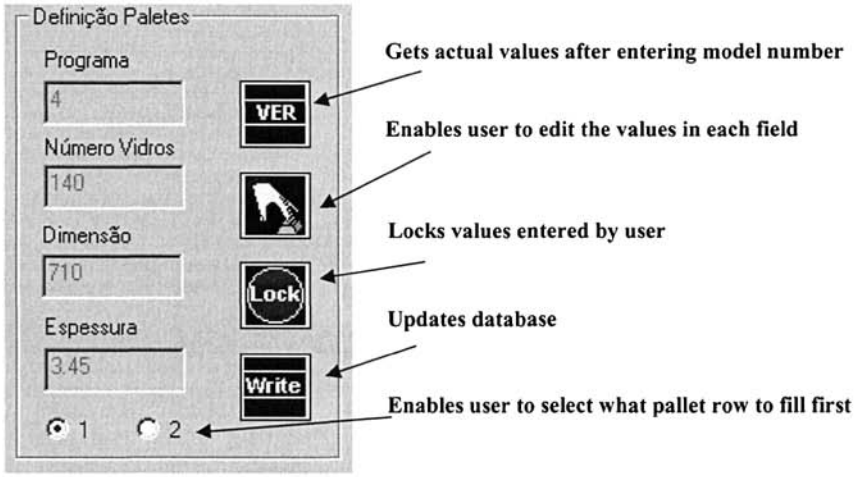
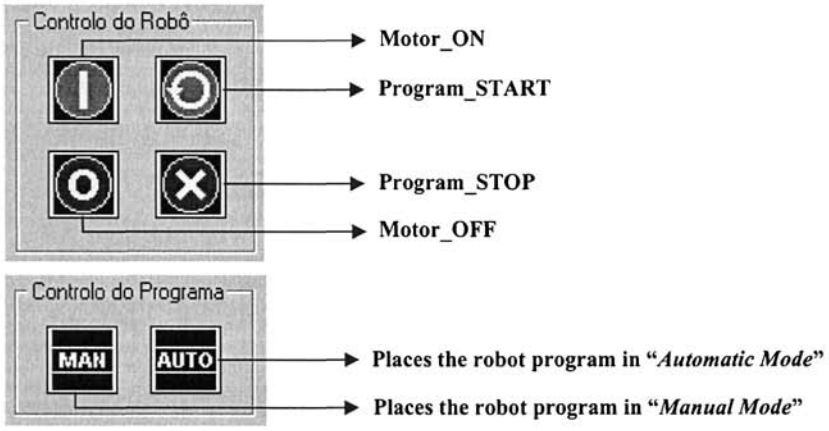


Figure 5.21 Accessing the database

Figure 5.21 shows the place where the user can change the glass model definition database. This operation is only possible, nevertheless, when the robot is waiting for operating mode selection. This procedure was implemented done for safety reasons, in a way to avoid corrupting the working database.



Example: Manual mode commanding routine (Visual C++ :NET 2003)

```
void CFornoDlg::Onmanual()
{float valor;
  fprintf(log,"%s - Comando de MANUAL.\n",tbuffer);
  if (m_pon.InitClient("babylon",5) >= 0)
  {valor=1236;
   nresult = m_pon.WriteNum("decision1",&valor);
   if (nresult <0) {m_log.SetWindowText("Error in the MANUAL command.");
```

```
   fprintf(log,"%s - Error in the MANUAL command.\n",tbuffer);erro=1;
m_erro.ShowWindow(1);}
   else m_log.SetWindowText("MANUAL command.");
    m_pon.DestroyClient();
  } else
  {m_log.SetWindowText("Robot didn't answer ... operation cancelled.");
   m_comms.SetIcon(AfxGetApp()->LoadIcon(IDI_smile2));
   m_erro.ShowWindow(1);
  }
}
```

Figure 5.22 Controlling the robot program

As already mentioned, commanding automatic or manual mode means accessing to a different set of functionalities. This operating mode change procedure is implemented in RAPID (ABB programming language) with the following simplified code (database access removed for simplicity):

```
WHILE never_end=FALSE DO
  WaitUntil (decision1=1235) OR (decision1=1236)\MaxTime:=1\TimeFlag:=timeout;
  IF timeout=TRUE THEN
  ENDIF
  IF (decision1=1235) THEN
    auto_mode;          <-------- Module that implements the "Automatic Mode"
    decision1:=0;
  ENDIF
  IF (decision1=1236) THEN
    manual_mode;        <-------- Module the implements the "Manual Mode"
    decision1:=0;
  ENDIF
ENDWHILE
```

5.4.3 On-line monitoring

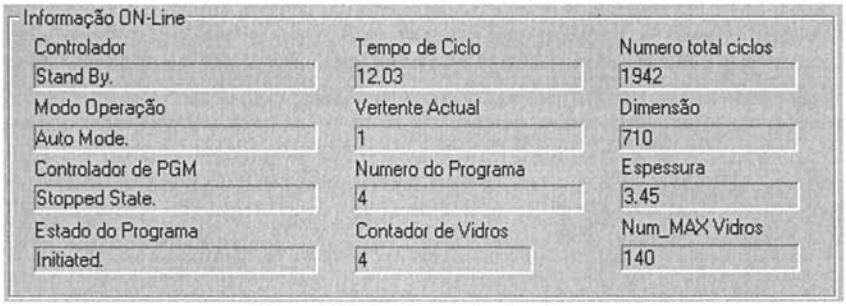
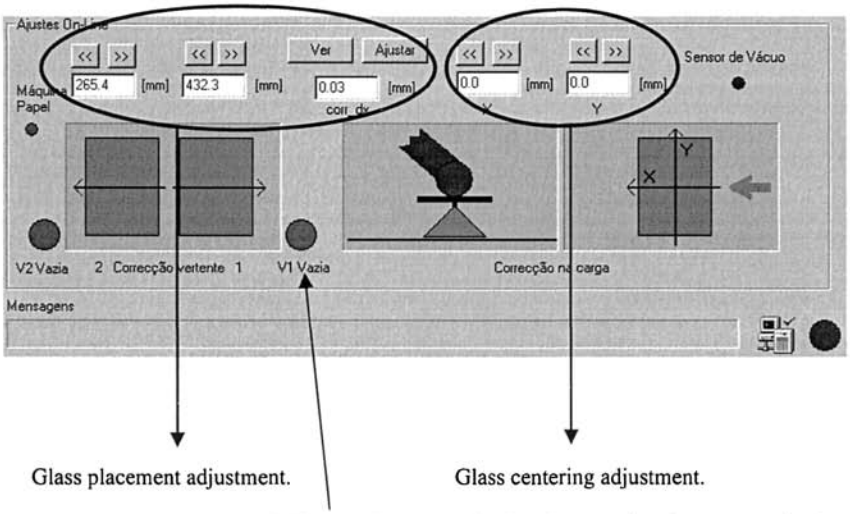


Figure 5.23 Online monitoring data

This feature (Figure 5.23) allows the user to quickly observe production data, such as: model in use, pallet row in use, number of cycles (pieces) performed since the

last counter erase, number of glasses palletized in the current pallet, last cycle time, robot working modes, and so on. This information is obtained directly from the robot, making monitoring calls to the relevant services. These calls are triggered by a timer interrupt routine, programmed to monitor the system in cycles of five seconds. A complete cycle, i.e., the operation of picking and palletizing a glass, takes about nine seconds, which justifies the polling monitoring option and the choice of a monitoring cycle of five seconds.



Note – The green and red indicators show permitted and error situations, respectively. Consequently, when a red indicator is present, the operator should interpret the warning and act accordingly.

Figure 5.24 – Adjusting online

Many times, due to operational difficulties in the production line, or centering errors, etc., it is necessary to make small adjustments in the palletizing process without stopping production. The operator may perform those adjustments using only a mouse (Figure 5.24), observe results, and correct the problem without stopping production. This type of procedure is fundamental for production environments characterized by high production rates and very tight quality control, as is the case of the automobile components industry.

Finally, another important operation under "*Automatic Mode*" is the operation of measuring the pallet parameters. That is done, as already mentioned, when a new empty pallet is introduced. This measurement must be done in every pallet, since they differ from each other significantly. Without this procedure, the palletizing process would fail. The robot is commanded to extend the precision contact sensors and use them to measure the pallet parameters. The robot uses three contact sensors, placed in the vertices of a triangle, to orient itself parallel to each surface and compute the angles around the robot's world reference system (Figure 5.25).

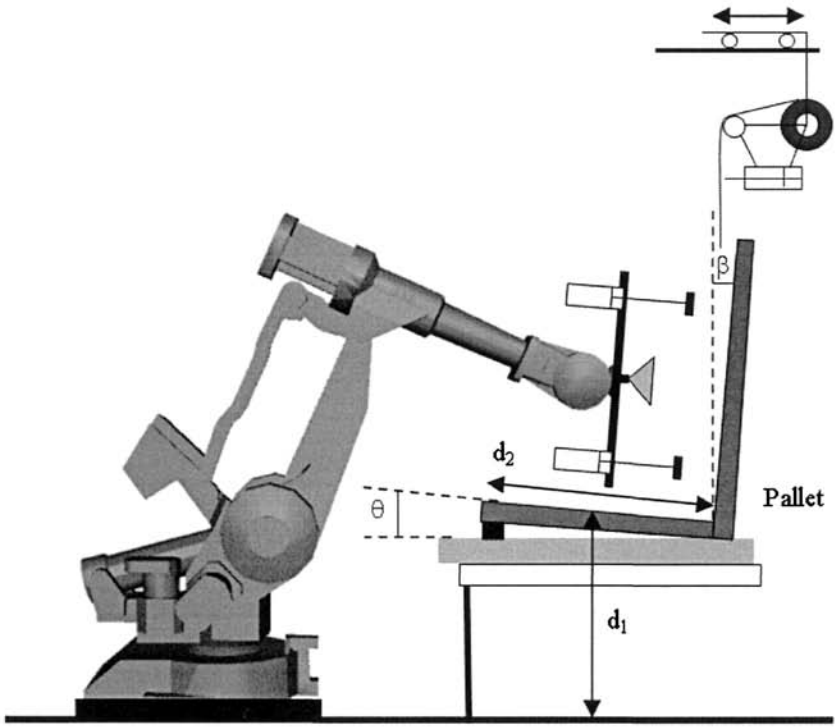


Figure 5.25 Getting pallet parameters: d_1, d_2, θ and β

The routine associated with this process is very simple and is presented below in a simplified form:

```
PROC check_pal()
  WaitUntil (divazia1=0) AND (divazia2=0)\MaxTime:=5\TimeFlag:=timeout;
  IF timeout=TRUE THEN
    TPWrite "Pallet not empty ...";
    PulseDO doerros;
    EXIT;
  ENDIF
  MoveJ pal_app,velocity,z100,toolt;
  sensores_on;
  MoveL RelTool(pal_up,0,0,250),velocity_app,fine,toolt;

  // Angle of the back of the pallet with the vertical axis

  SearchL\PStop,disen1,temp,RelTool(pal_up,0,0,500),velocity_search,toolt;
  MoveL temp,v10,fine,toolt;
  temp:=CRobT(\Tool:=tool_sen1);
  WHILE (disen2=0) AND ((disen3=0)) DO
```

Empty pallet??

Contact sensors in position

```
  MoveJ RelTool(temp,0,0,0\Ry:=-0.1),velocity_search,fine,tool_sen1;
  temp:=CRobT(\Tool:=tool_sen1);
 ENDWHILE
 pal_actual:=CRobT(\Tool:=toolt);
 angle1:=Abs(90-Abs(EulerZYX(\Y,pal_actual.rot)));
 TPWrite "Back Angle = "\Num:=angle1;

 // Angle of the base of the pallet with the horizontal axis

 MoveJ pal_up,velocity_app,fine,toolt;
 MoveJ pal_down,velocity_app,fine,toolt;
 SearchL\PStop,disen1,temp,RelTool(pal_down,0,0,500),velocity_search,toolt;
 MoveL temp,v10,fine,toolt;
 temp:=CRobT(\Tool:=tool_sen1);
 WHILE (disen2=0) AND ((disen3=0)) DO
  MoveJ RelTool(temp,0,0,0\Ry:=-0.1),velocity_search,fine,tool_sen1;
  temp:=CRobT(\Tool:=tool_sen1);
 ENDWHILE
 WaitTime 0.2;
 temp:=CRobT(\Tool:=toolt);
 angle:=Abs(EulerZYX(\Y,temp.rot));
 TPWrite "Base Angle "\Num:=angle;
 temp1:=RelTool(pal_actual,-(dim{modelo}/2-(pal_actual.trans.z-temp.trans.z)),0,0);
 pal_actual:=temp1;
 MoveJ pal_down,velocity_app,z50,toolt;
 MoveJ pal_app,velocity,z100,toolt;
 sensores_off;
ENDPROC
```

Height and dimension of the pallet

Retract contact sensors

5.4.4 Discussion and Results

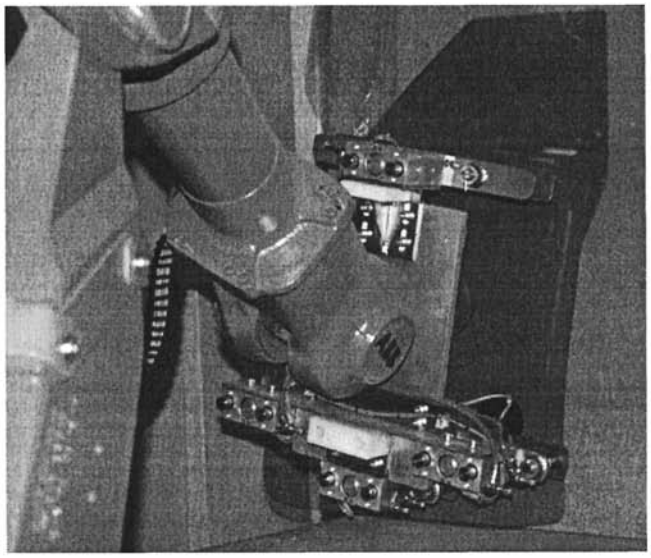
The system (Figure 5.26) presented in this section is a good example of a flexible robotic industrial system, capable of handling any production situation. The system relies on operator command and judgment, enabling him to fully parameterize production and introduce new production models. Besides of that, the operator may also introduce adjustments and change working conditions online, without stopping production, which is a powerful tool to handle production variations and difficulties. These features were obtained just by implementing a collection of services capable of handling all the anticipated production requirements, exposing them to the remote computer (*client*) where the operator interface is implemented. In this way, production may be tailored in a very flexible way, enabling the operator to solve virtually any operational situation.

Operational results are promising:

- Operators adapted easily to the system, which is always a good result considering their average skills
- Achieved production cycle is of aboutnine seconds per glass, which is more than is required

- The pallet measuring procedure takes about 25 seconds to complete, which is compensated by the very fast cycle time. The average overhead introduced by this procedure in the cycle time is about 25/280 = 0,089 ~ 0,1s (taking an average number of 280 glasses per pallet), which has no meaning
- The system works 24 hours a day without any need for operator supervision

It is worthwhile to point out that this system uses a client-server architecture, explained elsewhere [2-5] (see Section 3.2), developed to be used with robotic cells. Using this architecture implies the clear intention to distribute functions to all "*intelligent*" components of the robotic cell, leaving to the central PC (*the client*) the tasks of making the service request calls, properly parameterized, and displaying system information to the user. The PC is the user's commanding interface, and his window to the system. The developed software was built from scratch and the authors didn't use any commercial software, apart from operating systems (for example, *ABB Baseware 4.0* for the industrial robots, and *Microsoft Windows 2000* with *Service Pack 4* for the PC) and developing tools (*Visual C++ .NET 2003* [12] from *Microsoft*). A port of the *SUNRPC 4.0* [9] package for *Windows NT/2000/Xp*, a free open package originally developed for *UNIX* systems, was also used. The porting effort was, nevertheless, completely done by the author.



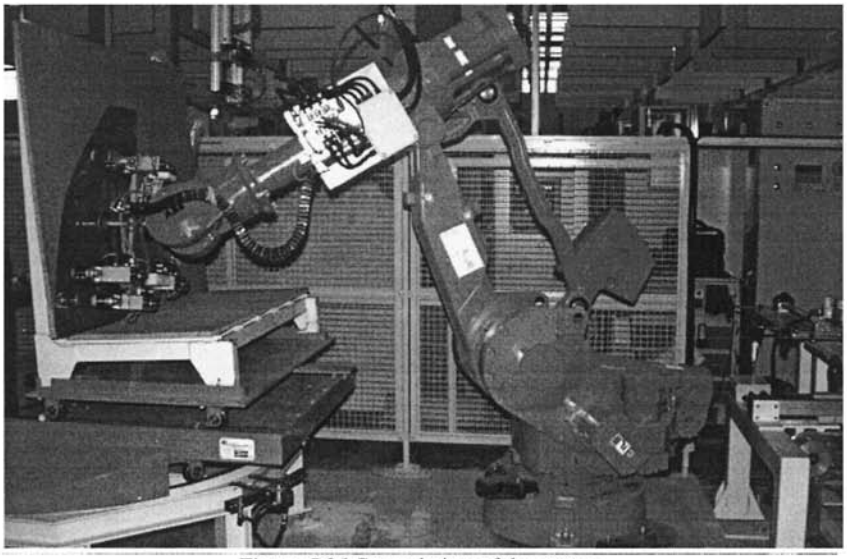


Figure 5.26 General view of the system

5.4.5 Conclusion

The system presented in this section is an implementation of a distributed software architecture developed to work with industrial robotic cells. The main objective was to be able to change production conditions online, and make adjustments to the working parameters so as to cope with production variations. The system was presented in some detail, giving special attention to the software designed to parameterize, monitor, and adjust the production setup enabling online adjustments to the working conditions. Obtained operational results demonstrate the interest of these types of systems for multi-model production environments, where high production rates and quality demands are a key factor. Finally, the obtained system is also a good example of man-machine cooperation, demonstrating the advantages of mixing human and automatic labor in actual manufacturing plants.

5.5 References

[1] ABB Robotics, "IRB6400 User and System Manual", ABB Robotics, Vasteras, 2002.
[2] Pires JN, Sá da Costa JMG, "Object Oriented and Distributed Approach for Programming Robotic Manufacturing Cells", IFAC Journal on Robotics and Computer Integrated Manufacturing, February 2000.
[3] Pires, JN, "Complete Robotic Inspection Line using PC based Control, Supervision and Parameterization Software", Elsevier and IFAC Journal Robotics and Computer Integrated Manufacturing, Volume 20, N.1, 2004.

[4] Pires, JN, Paulo, S, "High-efficient de-palletizing system for the non-flat ceramic industry", Proceedings of the 2003 IEEE International Conference on Robotics and Automation, Taipei, 2003.
[5] Pires, JN, "Object-oriented and distributed programming of robotic and automation equipment", Industrial Robot, An International Journal, MCB University Press, July 2000.
[6] Pires JN, "Interfacing Robotic and Automation Equipment with Matlab", IEEE Robotics and Automation Magazine, September 2000.
[7] Pires, JN, Godinho, T, Ferreira, P, "CAD interface for automatic robot welding programming", Sensor Review Journal, MCB University Press, July 2002.
[8] Pires, JN, and Loureiro, Altino et al, "Welding Robots", IEEE Robotics and Automation Magazine, June, 2003
[9] Bloomer J., "Power Programming with RPC", O'Reilly & Associates, Inc., 1992.
[10] Box D., "*Essential COM*", Addison-Wesley, 1998
[11] Rogerson D., "*Inside COM*", Microsoft Press, 1997.
[12] Visual C++ .NET 2003/2005 Programmers Reference, Microsoft, 2003 (reference can be found at Microsoft's web site in the Visual C++ .NET location)

6

Final Notes

6.1 Introduction

Dear reader, I hope you had fun reading and exploring this book, because in my opinion that is a fundamental outcome of a technical book. Furthermore, a book about robotics and automation must stimulate the reader curiosity and interest to explore further on its own.

This book is a practical guide about industrial robotics and related subjects. My primary objective was to introduce you to the fantastic world of robotics and ride with you through ideas, examples, and industrial solutions showing how things can be done, what are the available alternatives and challenges. Robotics and automation is a multidisciplinary subject that calls for creativity and innovation. It poses a permanent challenge for performance and practical results and consequently is a perfect subject for inventive and dedicated people, for whom this book was written. For that reason, the book presents a considerable amount of examples and solutions, allowing readers to see, from time-to-time, the complete picture of building a robotic manufacturing system, which constitutes also an invitation to maintain the focus. That is important. Robotics is an interesting subject and people are naturally attracted by its applications and achievements. Nevertheless, due to its multidisciplinary nature, robotics is also a very demanding field requiring knowledge of physics, electronics, mechanics, computer science, and engineering. Consequently, a book in the field gains by having examples and practical implementations. That was the "*design option*" followed when planning and writing the book. You can find the code of several of the presented examples along with pictures, videos, and other material at:

http://robotics.dem.uc.pt/indrobprog

The access to the site is restricted and requires a login "*username*" and "*password*". Visit the web site for details on how to obtain a valid login. As author

of this book, I'll keep the website updated so that it is a good source of information on:

- New developments
- Interesting solutions
- Interesting scientific and technical papers
- Interesting books
- Industrial trends in terms of technology

Most of these issues are related to new developments that result from R&D projects done in universities, research institutes, and companies, or in cooperation between academia and industry, resulting in technical papers, books, and new products. Robotics and automation is perhaps one of the most interesting cases of industry-academia cooperation since most of the developments require scientific, technical, and operational advances from both worlds to reach higher levels in terms of manufacturing flexibility and agility.

To be faithful to the basic "*design option*" adopted in this book, we will finish with another example. This final case is about a technical solution designed to reconfigure an old industrial robot, making it accessible through a local area network (LAN), and allowing programmers and system engineers to offer remote services to users.

6.2 Operation "*Albert*"

Albert is the name of an old robot that we acquired for our laboratory (Figure 6.1). The primary objective behind the acquisition was to obtain a nice industrial machine dedicated to teaching activities and to be included in laboratory classes of the discipline of "*Industrial Robotics*" (4th year of the Mechanical Engineering course). *Albert* worked for a few years in industry doing several types of tasks: manipulation, gluing, and labeling. After retiring from industry it is now starting a promising career in academia. Technically, *Albert* is an anthropomorphic robot manipulator (from 1992, build year) manufactured by *ABB Robotics* (model IRB1500) and equipped with an ABB S3 robot controller [1], i.e., it is a robot from 1992 but carrying technology from the mid eighties. Consequently, it is a rather old system with the following basic characteristics:

- Anthropomorphic manipulator (model ABB IRB1500): 5kg of payload, 6 axis, 0.1mm of repeatability and a fairly interesting workspace area (~1400mm)
- ABB S3 robot controller: This is the main disadvantage of *Albert*, since the S3 system is old and not carrying much of the interfaces required by actual industrial manufacturing systems. The controller is programmed using the programming language *ARLA* and has 16 digital inputs, 16

digital outputs, a serial port for data communication, and a very basic teach pendant (Figure 6.2).

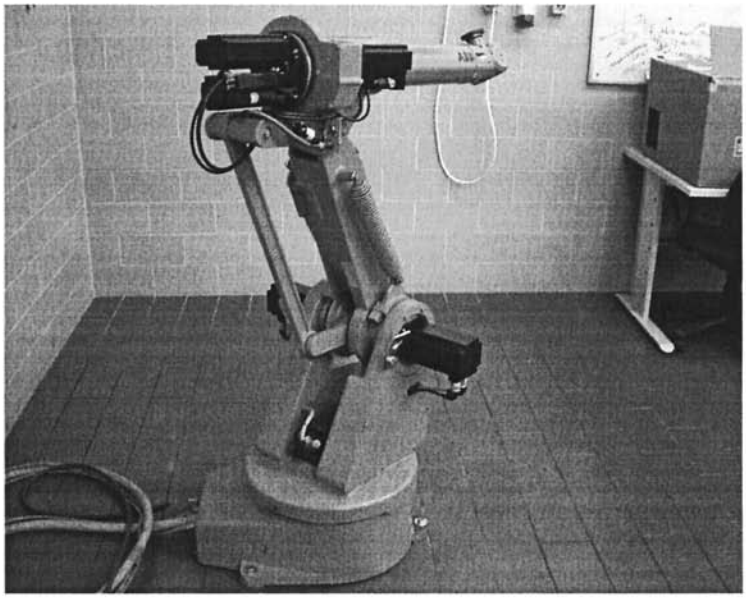


Figure 6.1 *Albert* is an ABB IRB1500 manipulator

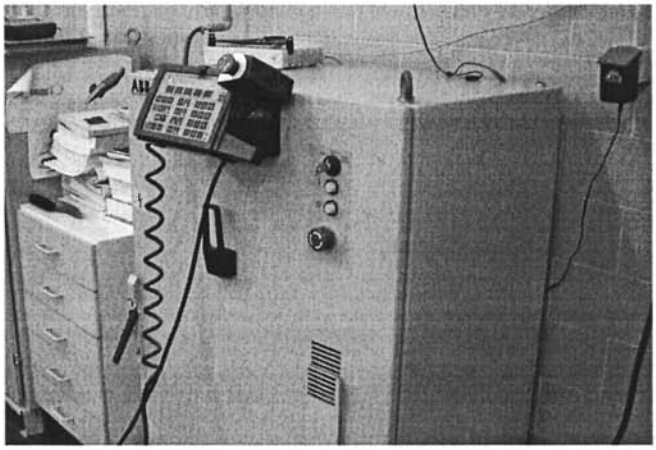


Figure 6.2 S3 robot controller

Consequently, this is mechanically a very interesting machine, very similar to its successor, the IRB1400 model. In fact, they share the same wrist design, which gives to the arm an excellent maneuverability. Nevertheless, because it is an old

system with very deficient communication interfaces, without any LAN interface, an old programming language (although sufficiently powerful) and a very basic user interface, *Albert* needs to be upgraded to be useful for teaching and training tasks.

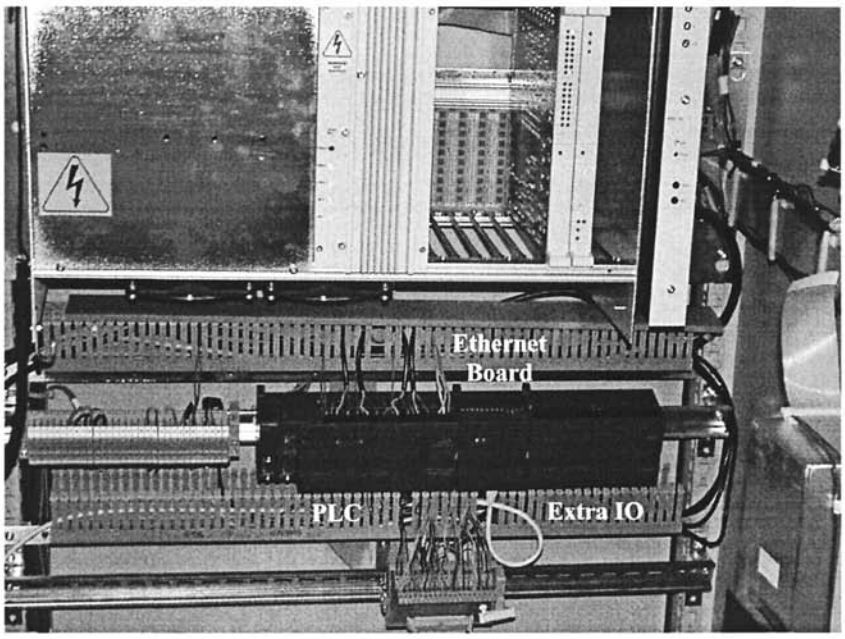


Figure 6.3 S3 cabinet with the extra hardware

To provide the system with a LAN interface, and the ability to offer programmed services to remote clients, while keeping the available system functionalities, the following actions were performed to upgrade the old *Albert* (Figure 6.3):

- A PLC (*S7-266* from *Siemens*) was added to the system, connected to the robot using the IO digital boards available in the S3 system. Consequently, a very simple parallel interface was added to transfer data between the PLC and the robot controller
- An *Ethernet* board (*CP 243-1* from *Siemens*) was also added to the system, connected to the PLC, to enable the system to interface with the LAN available in the laboratory. Consequently, remote users interface with the robot controller through the PLC, which means that a basic data protocol must be defined to exchange information between remote users and the running robot programs. That is a very simple task and was already used in Chapter 3
- An extra IO module was also added to the PLC to provide a supplemental set of IO digital line to use with applications.

The PLC is accessed using the *Ethernet* board and a simple UDP messaging system. To simplify the access, we used the *Siemens S7-200 OPC Data Access (OPC DA) Server* for the S7-200 (a server that is part of the *Siemens S7-200 PC Access* package). This server provides a means to access the PLC memory allowing the user to execute read/write operations on the entire PLC memory spaces (includes program variables, IO variables, special memory bits, etc.).

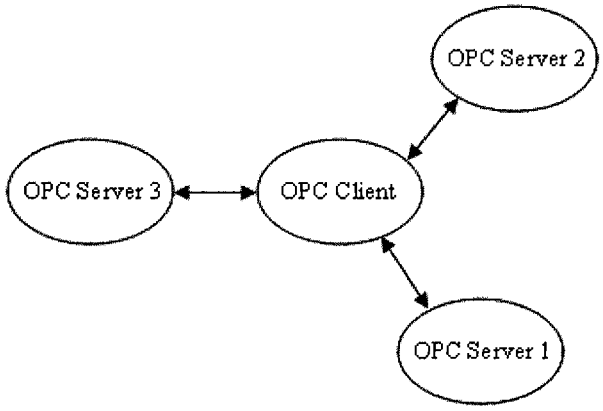


Figure 6.4 OPC client-server connection

Basically, OPC (*OLE for Process Automation*) [2, 3] was designed to allow client applications to access data from *shop floor* devices in a consistent and transparent way. Therefore, the OPC client applications interface with software modules (the OPC servers) and not with the hardware directly. This means that they rely on software components provided by the hardware manufacturer to efficiently access and explore the hardware features. Consequently, changes and hardware upgrades will not affect the user applications.

With OPC, whose specifications [3] include a set of *custom COM interfaces* [4] (used when building client applications) and a collection of *OLE automation interfaces* [5] to support clients built using high-level languages and applications (*Visual Basic* and *Excel*, for example), users can take advantage of the nice features of DCOM to facilitate client access to the system features. An OPC client can connect to OPC servers provided by any vendor that followed the OPC specification [3] (Figure 6.4).

Basically there are three types of OPC servers [2, 6]:

1. *OPC Data Access Servers (OPC DA Servers)* – This type of server is used to offer read/write data services to the client application. OPC DA servers constitute a powerful and efficient way to access automation and process control devices

2. *OPC Alarm and Event Handling Servers (OPC AE)* – This type of server is used to implement alarm and event notifications services to be used with client applications

3. *OPC Historical Data Access Servers (OPC HDA)* – This type of server is used to access (read/write) data from an historian engine

In this project to upgrade and reconfigure *Albert* an OPC DA server [7] is used to access the PLC. An OPC DA client application designed to access the PLC resources needs to deal with three types of objects:

1. *OPC DA Servers* – maintains information about the server and operates as a group container

2. *OPC DA Groups* – provides the mechanisms for containing and organizing items. Every OPC group has a particular update rate that must be set by the OPC client

3. *OPC DA Items* – the items are the real connections to the system resources. An item could represent a bit (like a memory bit or IO digital signal, etc.), a byte, a word, etc

Consequently, to access data from the hardware resource through the OPC server, the client should follow the following procedure:

- Connect to the OPC server
- Create an OPC group to perform synchronous reads/write operations
- Add the necessary items to the group
- Monitor the actual state of the items, or make asynchronous read/write operations

With *Albert,* twelve digital IO inputs and twelve digital IO outputs are used as data bus for robot-PLC communication. Some of those IO lines will be use to control the information flow between the robot and PLC. The remaining four digital inputs and four digital outputs will be used for special operations (Table 6.1).

To demonstrate how this can be used to command *Albert* from a remote PC, consider that the robot "knows" five positions, which are available for user request. The idea is to build a simple OPC client application to set up an OPC connection to the *Siemens S7-200 OPC Server*, and implement the necessary actions to command the robot to move to the user-selected positions.

Table 6.1 IO assignment for robot-PLC communication

Robot	**PLC**	**Description**
DI1 to DI12	Q0.0 to Q1.3	*Data IN*
DO1 to DO12	I0.0 to I1.3	*Data OUT*
DI13	Q1.4	*Motor_ON*
DI14	Q1.5	*Motor_OFF*
DI15	Q1.6	*Program_RUN*
DI16	Q1.7	*Program_STOP*
DO13	I1.4	*Motor_State*
DO14	I1.5	*Program_State*
DO15	I1.6	*System_State*
DO16	I1.7	*Emergency_State*

With that objective in mind, the following items were defined in the OPC server (Figure 6.5):

q0 – byte that contains the digital outputs Q0.0 to Q0.7
q1 - byte that contains the digital outputs Q1.0 to Q1.7
i0 - byte that contains the digital inputs i0.0 to i0.7
i1 - byte that contains the digital inputs i1.0 to i1.7

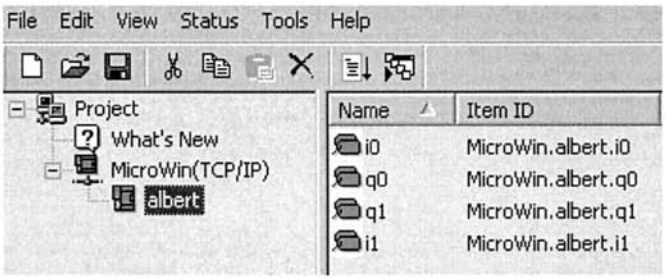


Figure 6.5 Items defined in the OPC server for this simple example

To implement the possibility of moving the robot using the OPC server, the following sequence is adopted:

- The robot waits for Q0.7 = DI8 = 1; means that a valid command is ready
- The commanded position is specified through bits Q0.0 (DI1) to Q0.4 (DI5), i.e., Q0.0 (DI1) is associated with P1, Q0.1 (DI2) with P2, …, Q0.4 (DI5) with P5
- The robot program jumps to "MOVE P1" routine and acknowledges the received command by making DO8 = I0.7 = 1
- The commanding PC should confirm the motion just by making q0 = DI1-DI8 = 0
- Robot makes DO8 = I0.7 = 0 and moves to the commanded position.
- Robot program jumps to the beginning and waits for a new command

Consequently, the program running on the robot controller (coded using *ARLA*) looks like the generic code presented in Figure 6.6.

```
while never_end;
  wait DI8 = 1;
  switch (byte DI1-DI8)
    case 1: DO8 = 1; wait (word DI1-DI8) = 0; DO8 = 0; Move P1;
    case 2: DO8 = 1; wait (word DI1-DI8) = 0; DO8 = 0; Move P2;
    case 4: DO8 = 1; wait (word DI1-DI8) = 0; DO8 = 0; Move P3;
    case 8: DO8 = 1; wait (word DI1-DI8) = 0; DO8 = 0; Move P4;
    case 16: DO8 = 1; wait (word DI1-DI8) = 0; DO8 = 0; Move P5;
  endswitch;
endwhile;
```

Figure 6.6 – Generic code running on *Albert's* controller

The OPC client application designed to connect to the OPC server, monitor the selected items and interface with the PLC (and through it to the robot controller) is represented in Figure 6.7.

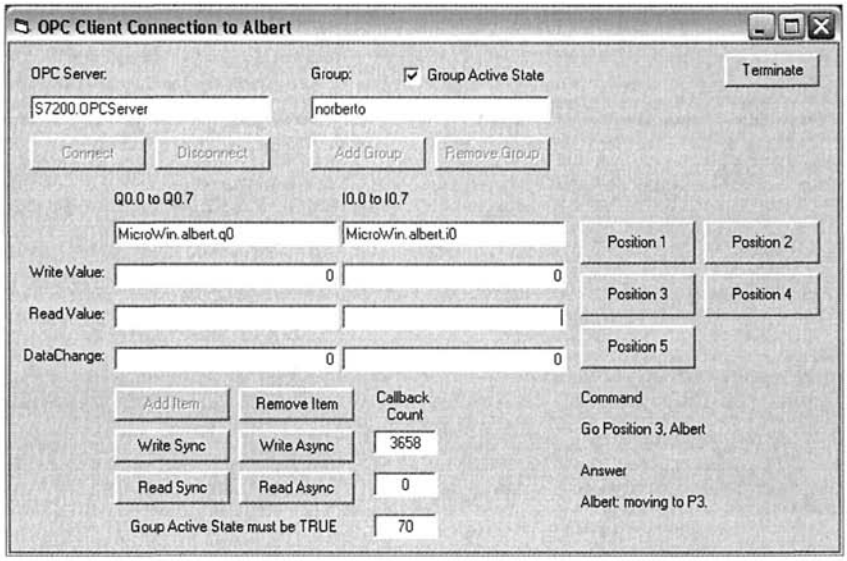


Figure 6.7 OPC client application designed to command the robot

The client application creates a group named "*norberto*" and enables the user to add the items of interest. In this, case the selected items are *Microwin.albert.q0* and *Microwin.albert.i0*. The default group updated rate is 100ms.

When a command is selected (using the software buttons "*Position 1*" to "*Position 5*"), the client application follows the above sequence just by monitoring the robot

response (through the PLC interface), and acting accordingly. Figure 6.8 reveals the code associated with the action of commanding the robot to move to P1.

```
Private Sub p1_Click()
  If txtChangeVal(1).Text = "0" Then
    txtWriteVal1.Text = "129"      <-- Command valid + MOVE to P1
    lp1 = 1
    cmdWriteAsync                  <-- Call to WriteAsynchronous
    cmd_sent.Caption = "Go Position 1, Albert"
  Else
    cmd_sent.Caption = "Albert: I'm not ready!"
  End If
End Sub

Private Sub Timer1_Timer()
 If (lp1 = 1) Then
  If (txtChangeVal(1).Text) = "128" Then   <-- Robot received the command
    txtWriteVal1.Text = "0"
    cmdWriteAsync                          <-- Call to WriteAsynchronous
    lp1 = 0
    answer.Caption = "Albert: moving to P1."
  End If
 End If
...
 If (lp5 = 1) Then
  If (txtChangeVal(1).Text) = "128" Then
    txtWriteVal1.Text = "0"
    cmdWriteAsync                          <-- Call to WriteAsynchronous
    lp2 = 0
    answer.Caption = "Albert: moving to P5."
  End If
 End If
End Sub
```

Figure 6.8 Code associated with the command action move to P1

This example shows clearly the usefulness of the updated *Albert* for teaching and training tasks. In the update process a PLC was added to the robot controller cabinet, including an extra IO board and an *Ethernet* card (on the PLC bus), which can work in parallel with the application running on the robot controller. These new features can be explored when building applications, and since the user needs to deal with the robot controller software, the PLC software, and the protocol to manage the robot-PLC communication (as shown in the presented example), it is fair to say that the new *Albert* constitutes a very nice platform to learn about robotics and automation.

6.2.1 And "*Albert*" Speaks

From the material presented in Chapter 4, the task of adding a speech interface to *Albert* is straightforward. Nevertheless, it will be done in this section, step-by-step,

because in the process a few details about the human-robot interface will be further clarified. For simplicity, we'll use the same setup presented above.

The first thing to decide is the structure of the voice commands. The best option is the "*command and control mode*" (see Section 4.2.3) because it is more adapted to industrial situations that require a clear and safe identification of commands. With this operation mode, the software needs to identify the sequence of words and strings that compose the command, and generate the appropriate action to the robot controller. Consequently the selected command structure is

name_of_machine command parameters

where "*name_of_machine*" is the name attributed to the machine (in our case "*Albert*" or "*robot*"), "*command*" is a word identifying the command and "*parameters*" are words or strings identifying the parameters associated with the particular command.

In the presented example, there are four commands available:

"*hello*" – enables the user to query if the interface is available
"*initiate*" – initiates the speech interface
"*terminate*" – suspends the speech interface
"*move*" – commands the robot to move to a position

These commands are associated to the machine "*Albert*" (or "*robot*"), which means that they are associated with the pre-command string "*Albert*" (or "*robot*").

The next step is to write the above defined grammar using a standard format that can be understood by our software. There are two ways to achieve that:

- Include grammar specific instructions in the body of the software (hard-coded grammar). This means that any change in the grammar structure, or a simple update in the command list, requires another compilation of the application software.
- Specify the grammar using XML files. This is straightforward and flexible to changes and updates.

In the presented example, an XML file is used to specify the grammar (Figure 6.9). Since we use English and Portuguese recognizers, two XML grammars were built to allow the user to select the language. The application reads the grammar from the XML file, selects the recognizer to use based on the language ID tag, commits the rules, and handles the recognition events. When a certain rule is identified, an event is fired by the recognition engine and catch by our application that executes the appropriate actions (Figure 6.10).

```
<GRAMMAR LANGID="409">
 <DEFINE>
  <ID NAME="test" VAL="1"/>
  <ID NAME="move" VAL="2"/>
  <ID NAME="position" VAL="3"/>
  <ID NAME="init" VAL="4"/>
 </DEFINE>
 <RULE NAME="ROOT" TOPLEVEL="ACTIVE">
  <L>
   <P>albert</P>
   <P>robot</P>
  </L>
  <RULEREF PROPNAME="move" PROPID="move" NAME="move"/>
  <P>to</P>
  <RULEREF PROPNAME="position" PROPID="position" NAME="position"/>
  <O>please</O>
 </RULE>
 <RULE NAME="START" TOPLEVEL="ACTIVE">
  <L>
   <P>albert</P>
   <P>robot</P>
  </L>
  <RULEREF PROPNAME="init" PROPID="init" NAME="init"/>
  <O>please</O>
 </RULE>
 <RULE NAME="move">
  <LN PROPNAME="move" PROPID="move">
   <PN VAL="1">move</PN>
   <PN VAL="2">go</PN>
  </LN>
 </RULE>
 <RULE NAME="init">
  <LN PROPNAME="init" PROPID="init">
   <PN VAL="1">initialize</PN>
   <PN VAL="2">terminate</PN>
   <PN VAL="3">hello</PN>
  </LN>
 </RULE>
 <RULE NAME="position">
  <LN PROPNAME="position" PROPID="position">
   <PN VAL="1">position one</PN>
   <PN VAL="2">position two</PN>
   <PN VAL="3">position three</PN>
   <PN VAL="4">position four</PN>
   <PN VAL="5">position five</PN>
  </LN>
 </RULE>
</GRAMMAR>
```

Figure 6.9 XML file containing the speech grammar (English version)

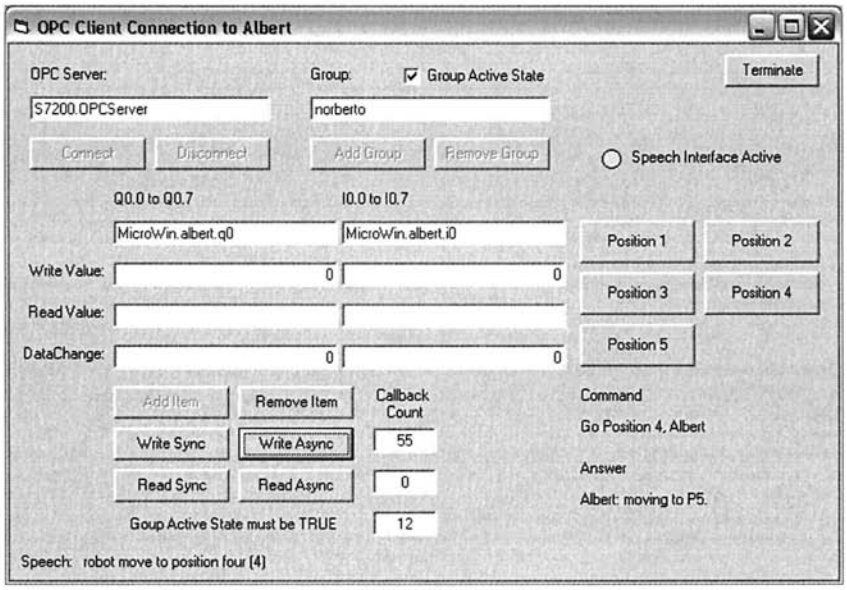
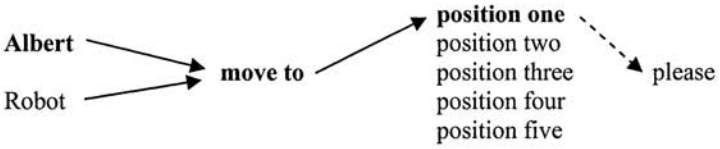


Figure 6.10 OPC client application with the speech interface included

Figure 6.11 show the code associated with the rules that command the robot to move to position one:




```
nprop = Result.PhraseInfo.Properties.Count
If nprop = 1 Then
 If Result.PhraseInfo.Properties(0).Children(0).Value = 1 Then
  answer.Caption = "initialize"
 End If
 If Result.PhraseInfo.Properties(0).Children(0).Value = 2 Then
  answer.Caption = "terminate"
 End If
 If Result.PhraseInfo.Properties(0).Children(0).Value = 3 Then
  answer.Caption = "hello"
  If Result.PhraseInfo.LanguageId = 1033 Then ------------> English
   Voice.Speak ("Hello, I am albert.")
  End If
  If Result.PhraseInfo.LanguageId = 2070 Then ------------> Portuguese
   Voice.Speak ("Olá, eu sou o alberto.")
```

```
  End If
 End If
End If
If (nprop = 2) Then
 If Result.PhraseInfo.Properties(1).Name = "position" Then
 If (Result.PhraseInfo.Properties(1).Children(0).Value = 1) Then
  speech_out.Caption = speech_out.Caption + " (1)"
  If Result.PhraseInfo.LanguageId = 1033 Then
   Voice.Speak ("Position one, master.")
  End If
  If Result.PhraseInfo.LanguageId = 2070 Then
   Voice.Speak ("Posição um, mestre.")
  End If
  p1_Click ──────────► Routine that commands the robot to move to P1
 End If
(...)
```

Figure 6.11 Visual Basic code associated with handling speech events: aspects related with the "move to position" command

When an event is received, the application needs to query the speech API for the property that was identified, and take the appropriate actions based on the returned values. It's a straightforward procedure based on the selected command structure defined in the XML file containing the speech grammar.

With this example, I finish this book. My sincere hope is that it could constitute a nice and useful resource of information and inspiration, but also a "*platform*" to stimulate your curiosity to proceed further in the area.

Because... Robotics is Fun!

6.3 References

[1] ABB Robotics, "IRB1500 Users and Systems Manual", ABB Robotics, Vasteras, 1992.
[2] Iwanitz, F., Lange, J., "OPC, Fundamentals, Implementation and Application", Huthig, 2nd edition, 2002.
[3] The OPC Foundation, http://www.opcfoundation.org
[4] Box D., "*Essential COM*", Addison-Wesley, 1998
[5] Rogerson D., "*Inside COM*", Microsoft Press, 1997.
[6] OPC Foundation, "OPC Overview", Version 1, OPC Foundation, 1998.
[7] Siemens Automation, "S7-2000 PC Access Users Manual", Siemens, 2005.

Index

Printed in the United States