SUPERVISORY COMPUTER CONTROL OF A FLEXIBLE MANUFACTURING CELL

By

Thomas Maxwell Kean

B. Eng. Memorial University of Newfoundland

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

MECHANICAL ENGINEERING

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

December 1988

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Mechanical Engineering

The University of British Columbia
Vancouver, Canada

Date Decenber 20, 1988

ABSTRACT

A combined software and hardware package has been developed which coordinates the activities of a Unimation PUMA 560 industrial robot and an ORAC CNC training lathe in a Flexible Manufacturing Cell environment under the supervision of a DEC VAX 11/750 computer. The hardware component is used to provide an intermediate interface between the VAX 11/750 software and the standard manual controls of the ORAC lathe. The interface to the PUMA robot is strictly software oriented and makes use of the Supervisor communication protocol provided by the robot's VAL II controller.

The software package has been designed as a hierarchial, multi-tasking system to facilitate modular development and a logical *division of labour*. The hierarchy consists of three main levels, with the individual machine controllers providing the lowest level. At the middle layer is a number of subprocesses which execute on the VAX 11/750 and are each responsible for the direct supervision of one machine controller. This supervision involves both sending commands and receiving status messages in the native language of the individual controllers. At the top layer of the hierarchy is a single process which is responsible for the overall coordination of the workcell activities.

Each subtask in the hierarchy communicates with its vertical neighbours through a series of communication protocols and command vocabularies developed for that purpose. Each subtask is also provided with a structured interface to the operator's console. This interface is implemented by a separate VAX process that provides a status window environment for each machine that is active in the cell, but also intercepts text messages and prompts so that they may be displayed to the operator one at a time.

Two different software packages have been developed for the top layer of the hierarchy. The ORCAM package is dedicated to the task of automatically manufacturing turned components from a CAD database. The FMC package provides a flexible interface to each of the supported machines and may be used to supervise any number of user-defined tasks.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to thank my academic supervisor, Dr. Farrokh Sassani of the Department of Mechanical Engineering, for providing the initial idea for this work. His assistance and patience during the completion of the work are gratefully acknowledged. I would also like to thank Dr. Dale Cherchas of the Department of Mechanical Engineering for his efforts in obtaining the VAL II upgrade for the PUMA 560 robot controller, and for his continual support of the VAX to PUMA interface.

Many thanks are also extended to the following people who helped make this project a reality. Mr. Robert Moore who wrote the ORACAP software, designed the tool changer, and provided many valuable suggestions. Mr. Alan Steeves who provided valuable knowledge on both the VAX 11/750 and the Zilog Z8 line of microcomputers. Mr. Graham Yarbourgh of Unimation Inc. who supplied a hand-written update to the VAL II manual prior to its formal release by the company. Mr. Bob Williamson of RSI Robotic Systems International Ltd. who graciously granted me a *working* Leave Of Absence to complete this project.

This work was funded in part by a Postgraduate Scholarship from the Natural Sciences and Engineering Research Council of Canada.

Final acknowledgement and thanks must go to my wife Arlene, to whom I dedicate this thesis. Her unending support and encouragement enabled me to complete this work.

Chapter 1

INTRODUCTION

Production automation is the engineering discipline concerned with the application of mechanical, electronic, and (more recently) computer based systems to the control and operation of manufacturing equipment [10]. This is a dynamic technology which was born during the industrial revolution (circa 1770), when simple production machines were designed to perform tasks previously done by manual labor. During the early stages of development the major emphasis was on increased productivity, which led to the evolution of manufacturing procedures and machines. In the last two decades, changing market trends coupled with a dramatic increase in the use of computerized controls in the form of CNC machining centers, industrial robots, and automatic guided vehicles, has caused a shift in emphasis to include greater flexibility as well as productivity.

1.1 Historical Perspective

Perhaps the most traditional symbol of automation is the mechanized flow line. Chronologically, this was the first major example of automated production to appear. It's origin can be traced to the work of Henry Ford in the manufacture of automobiles [10]. His advances in assemblyline mass-production techniques led to the development of

fixed-automation transfer lines. These lines consist of several machines or workstations each performing a specialized operation, with workpieces being transferred from station to station in a serial fashion. A well balanced transfer line is characterized by short production times and fast output, but also by extreme inflexibility to product alteration.

The level of automation utilized in early transfer lines was limited to plugboard controllers which could perform a fixed sequence of operations, and copying machines in which a stylus moved on a master part and produced a replica. The rapid growth of the aircraft industry in the late 1940's created an increased need for complicated parts with contoured surfaces. The high degree of accuracy required coupled with relatively small batch sizes made these parts difficult and costly to produce on conventional machines. In 1948 the U.S. Air Force started a research program that led to the development of the first Numerical Control (NC) milling machine by the MIT Servomechanism Laboratory in 1952 [11]. Simultaneous control of three axis was performed using numbers stored on punched binary tape as setpoints. This method of control provided far more accuracy than the previous plugboard control, removed the need to tie up the machine while making a master part, and introduced a level of flexibility suitable for medium volume production.

The expansion of digital computer technology in the 1960's led to the development of Direct Numerical Control (DNC) systems which were designed to replace hardware components typically duplicated in individual NC controllers with a sophisticated central computer. The most successful form of DNC has been the Behind-Tape-Reader (BTR) interface which leaves the conventional NC controller intact, except for the tape reader which is replaced by a telecommunication line to the DNC computer [10, 11]. Programs stored in the central computer are downloaded to individual NC machines upon demand.

The advent of minicomputers and microprocessors with their increased computational ability, smaller physical dimensions, and reduced costs, has largely nullified the driving force behind DNC systems. Current systems are characterized by an architecture known as Computer Numerical Control (CNC) in which much of the hardwired circuitry of early NC systems is replaced by a microprocessor imbedded directly into the controller of each machine.

Simultaneous with the advances in CNC systems was the development of the industrial robot. The definition of an industrial robot given by the Robotics Industries Association (RIA) is a "reprogrammable, multifunctional manipulator designed to move material, parts, tools, or special devices through variable programmed motions for the performance of a variety of tasks" [7]. The first commercial robot was manufactured in 1961 [11], but it was not until the late 1970's, when significant improvements in repeatability and accuracy were made, that industrial robots became commonplace in the manufacturing industry. Early robots were typically programmed by physically guiding them through the desired motion under manual control while positions were recorded for eventual play-back. As with NC machines, the ability to define positions in terms of numerical coordinates, without the use of the physical robot, greatly enhances the flexibility of modern robots, making them suitable for batch production.

Concurrent with the technological advances in NC machining and robotics were advances in manufacturing philosophy. Group technology is now widely used as a means of identifying and grouping components into part families to take advantage of similarities in design and manufacture. By arranging production equipment into manufacturing cells responsible for specific part families, a significant reduction in part handling may

be achieved. In contrast to the traditional process-type plant layout, other advantages are lower set-up times, less in-process inventory, and shorter lead times [10].

1.2 Current Research

The next logical step in the development of production automation has been the marriage of the group technology concept with the advanced robot and CNC systems. This marriage capitalizes on the reprogrammable nature of these systems to produce a manufacturing environment that's both productive and highly flexible. This structure uses a central computer to coordinate and monitor the activities of a number of individual machine controllers. The actual activities are generated on the central computer using off-line programming techniques and are not rigidly fixed as in hard-automation. This structure is known as a Flexible Manufacturing Cell (FMC) and has provided the backbone for an explosion of research into the factory of the future.

Recent advances in the areas of force control and robot accuracy have resulted in the development of a similar structure denoted as a Flexible Assembly Cell (FAC). The ultimate goal of production automation is the operation of a number of manufacturing and assembly cells under the supervision of a central computer. This architecture is known as a Flexible Manufacturing System (FMS) and must also incorporate a materials and tool handling system. Recent pilot studies into FMS have resulted in various degrees of successes and failures [9, 17, 14]. Analysis of these results are currently refining the structure of FMS so that it may serve as a vehicle for the parallel growth of CAD and CAM. This structure will pave the way for Computer Integrated Manufacturing

(CIM); the coordinated participation of computers in all phases of manufacturing: design, planning, manufacture, control, assembly, and testing.

1.3 Research Objective

Research into NC machining techniques started at UBC in the late 1970's with the purchase of two NC milling machines. This equipment was supplemented in 1983 with the purchase of an ORAC CNC training lathe and a PUMA 560 industrial robot. These acquisitions provided the impetus for two independent studies into off-line programming. The research into off-line programming of the ORAC lathe resulted in the ORACAP software package; an interactive programming package which produces ORAC NC code from a geometric description of the part [13]. The original research into off-line programming of robots culminated in an interactive package called AUTOP which automatically generates robot trajectories for a welding workstation [5]. Later research expanded the off-line programming capabilities to include optimized 2–D trajectories [6].

Although the ORACAP package includes an option for downloading programs to the ORAC lathe over a serial line, it does not provide any means for initiating or monitoring the actual machining process. The AUTOP package used to generate the welding trajectories does not provide any capabilities for communicating with the robot controller. In order to allow these off-line programming capabilities to be utilized in a manufacturing environment it was necessary to develop a means of communicating with, and controlling the activities of, each of the two machines.

This thesis describes the development of a supervisory control package for the VAX 11/750 computer that supports the integration of CAD data into an actual manufacturing

environment. This work had the specific objective of combining the PUMA 560 robot and the ORAC CNC lathe into a functioning FMC under the supervision of the VAX. Additional design requirements called for the flexibility to allow other manufacturing equipment to be added to the cell at a future date, and the modularity to support future use in the ongoing research into robotic assembly using force control.

Chapter 2

SYSTEM OVERVIEW

This chapter describes the overall structure of the supervisory control system developed for the VAX 11/750. The structure is introduced by giving some general guidelines from the literature that apply to the design of any manufacturing control environment. The specific factors affecting the VAX implementation are then discussed while outlining the chosen architecture. The details of the actual software and hardware required to interface and control the ORAC lathe and PUMA 560 robot are provided in later chapters.

2.1 General Structure of an FMC

The application of computers to the manufacturing industry has advanced at a phenomenal rate in the past twenty years, but it has not always been a pleasant experience. Many manufacturers have been left with nothing more than a high priced conversation piece as a result of their attempts to apply computer technology. A number of different architectures have been tried over the years with mixed results. Today, with the proliferation of small powerful computers, the hierarchial computer structure has become the

most effective and efficient arrangement for implementing computer control in manufacturing [10]. This structure supports a distributed control effort that results in balanced response times, and more manageable software.

The Flexible Manufacturing Cell is, by its very nature, a hierarchial computer system with at least two levels. The individual machine controllers represent the lowest level of the hierarchy and must be capable of supporting bidirectional communications with the higher levels. Both the machine and process states have to be available to the central computer on a continuous basis to allow logical decisions. The control effort at this level should be autonomous and each controller is ideally responsible for only one device or a group of closely related devices. This control effort, however, may require an awareness of the state of other machines and some horizontal communication may also be necessary at this level.

Despite the recent advances outlined in the preceding chapter, many NC machine tools and robots are still being developed primarily as stand-alone units. Depending on the level of intelligence embedded in each controller, an additional level of computer control may be required to provide the necessary communication capabilities. These interfaces should be designed such that the manual control and data entry functions remain operational and accessible in the event of a breakdown in the network.

Another factor affecting the number of hierarchial levels required in an FMC implementation is the number of machines being supervised. Unlike the machine controllers, computers at higher levels of the hierarchy are responsible for more than one device. To be truly productive and flexible, these higher levels must be organized to support the simultaneous operation of all machines. Numerous techniques, including prioritized interrupts and event-based multitasking, are available to support this function, but it is

essential that the computer's processing speed be matched to the rate at which events are likely to occur. If interrupt priorities are not balanced with respect to time spans, deadlocks may result and device synchronization may become impossible [9].

Counteracting the desire for well distributed control is the reality that each node must be capable of vertical communication with layers both above and below. Although local area networks (LAN's) have received a lot of attention lately, the task of linking the computers of different manufacturers is still a costly and questionable undertaking.

The prime consideration in computer based manufacturing control, be it FMC, FMS, or CIM, is flexibility. The key to this flexibility is a modular software design [14]. A modular design provides for gradual implementation and testing of components that will ultimately operate as a total system. By using formalized procedures for the sharing and transportation of data, new modules may be added with minimal disruption to the existing system. A modular design also allows the software component to take advantage of the same hierarchial structure as applied to the hardware component.

The design of manufacturing equipment is still very individualistic and various levels of status information and command capabilities are provided. Recent efforts to establish the Manufacturing Automation Protocol (MAP) developed by GM as an industry standard for communications among manufacturing equipment have not met with widespread acceptance. Until a standard is accepted, intermediate software translators (referred to as data pumps by Lambourne [12]) must be used to integrate these machines into the FMC environment. These modules represent an intermediate layer in the supervisory control system; communicating with the low level machine controllers on one hand, and the cell management software on the other. By developing a standard protocol for the

interface to the cell management level, the addition of future machines is simplified and added flexibility is gained.

The generic supervisory control system can therefore be seen to operate on three levels; high or supervisory level, middle or machine translation level, and low or hardware control level. Each level performs a specific task based on information received from adjacent levels. Communications between individual levels are governed by specific protocols that specify both the format and content of the information communicated. In general, commands are directed from top to bottom, while status information flows upward. Communication rates are highest at the lowest level, and diminish as one moves up the hierarchy.

The system also has a monitoring function that is not restricted to any one level. Software at all levels of the hierarchy collect data that is of importance to the system operator, and may be required to make decisions based on information that only the operator can supply [14]. The design of the operator interface is therefore not a trivial matter. If all operator interactions are funnelled through one layer the logical confinement of data within modules is lost, and the protocols for vertical communications become overburdened. However, if all levels communicate directly with the operator, some means of controlling the order and placement of messages must be implemented.

2.2 Design of Generic VAX/Machine Interface

One of the first tasks in designing any hierarchial software system is the logical *division of labour*. The distribution of tasks in the VAX implementation was directly affected by the requirement to support a number of different workcell objectives; machining,

assembly, and welding. This requirement dictated that the characteristics of individual machines should be *hidden* from the cell management software. By confining the majority of the monitoring and control to the middle layer, the supervisory software is presented with a simplified, more uniform status interface. This design streamlines the creation of new cell management packages by reducing the amount of software that would have to be duplicated in each package.

A uniform status reporting mechanism also simplifies the addition of new machines into existing cell management packages. Because all messages have the same syntax, a single module may be used to translate the status of any number of machines. Unfortunately, the command passing mechanism is not quite as simple. Most computer based equipment will likely have a file transfer feature, but the majority of functions will be highly dependant on the type of machine. Although some massaging of the command syntax may be performed by the middle layer software, a truly uniform interface to the cell management layer is prohibited. Consequently, each supervisory level package will have to be embodied with the appropriate command syntax for the new machine.

In addition to being independent of the cell management package being used, the software responsible for communicating with an individual machine should also be independent of the physical link to the machine. By writing this software as a device *driver*, the actual communication channel may be assigned at run time. This technique allows a machine to be relocated without affecting the operation of the software. It also allows the addition of an identical machine without requiring the duplication of software modules.

Once the modular structure is defined, the implementation must be addressed. As noted in the preceding section, the design of an FMC should support simultaneous control. This was not essential for the two-machine turning workcell under consideration, but was considered as a design objective to enhance future use.

The two techniques commonly used to handle asynchronous communications over a number of different data channels are interrupts and multitasking. If interrupts are to be used effectively, the number of interrupt priority levels must match the number of competing events. The VAX/VMS operating system supports user interrupts on all I/O channels through the Asynchronous System Trap (AST) routines, but all requests are assigned the same priority. Once an interrupt handler is entered the user process is said to be in AST mode and all other user interrupts for that process are blocked. User interrupts may also be blocked by the system during the execution of certain System Services, such as the Queue I/O With Wait (QIOW) routine which is used by the Fortran compiler to perform all read requests.

Due to the large number of I/O requests and other System Service calls required, the single priority user interrupt proved to be an ineffective means of implementing the FMC software, even for two machines. It was therefore necessary to implement the VAX portion of the supervisory control system as a multitasking hierarchy. This technique provides the best means of ensuring that each machine receives balanced attention. The VMS system uses a combination of assigned priorities, outstanding I/O requests, and time slicing to determine the amount of processing time that a given task receives. This combined precedence prevents a task that is waiting for an input message from blocking the execution of other *active* tasks, but also ensures prompt response once the required input is received.

The division of tasks utilized in the VAX-to-Machine interface is shown in Figure 2.1. Each separate process executing on the VAX is represented in the figure by a divided box with two labels. These two labels distinguish between the name of the process—the upper label—and the name of the software being executed by the process—the lower label. The process name is used both by the operating system and by other cooperating processes to determine the source of interprocess communications. Because a single software image can be executed within the context of any number of processes, the control of identical machines can be accomplished by running the driver under two different process names, as shown by the dashed lines in Figure 2.1. Each machine is therefore referenced by its process name (which must be unique) rather than by its brand name.

At the root of the multitasking hierarchy is the parent process which executes the cell management image. This software is responsible for the coordination of the overall activities of the cell. The distribution of commands and status information to and from the machine level is accomplished through a separate subprocess for each machine. These subprocesses are created and assigned names by the parent process as the need arises. The subprocess name, physical I/O channel, and driver image to be assigned to each machine are all read from a data file to facilitate modifications and additions.

At the present time there are two different cell management packages available, ORCAM and FMC. These two packages differ mainly in the way that the cell activities are determined. The ORCAM package determines the order and magnitude of these activities directly from a CAD database, while the FMC package expects the sequence of events to be defined by an external program. Both of these packages are described in detail in Chapter 5.

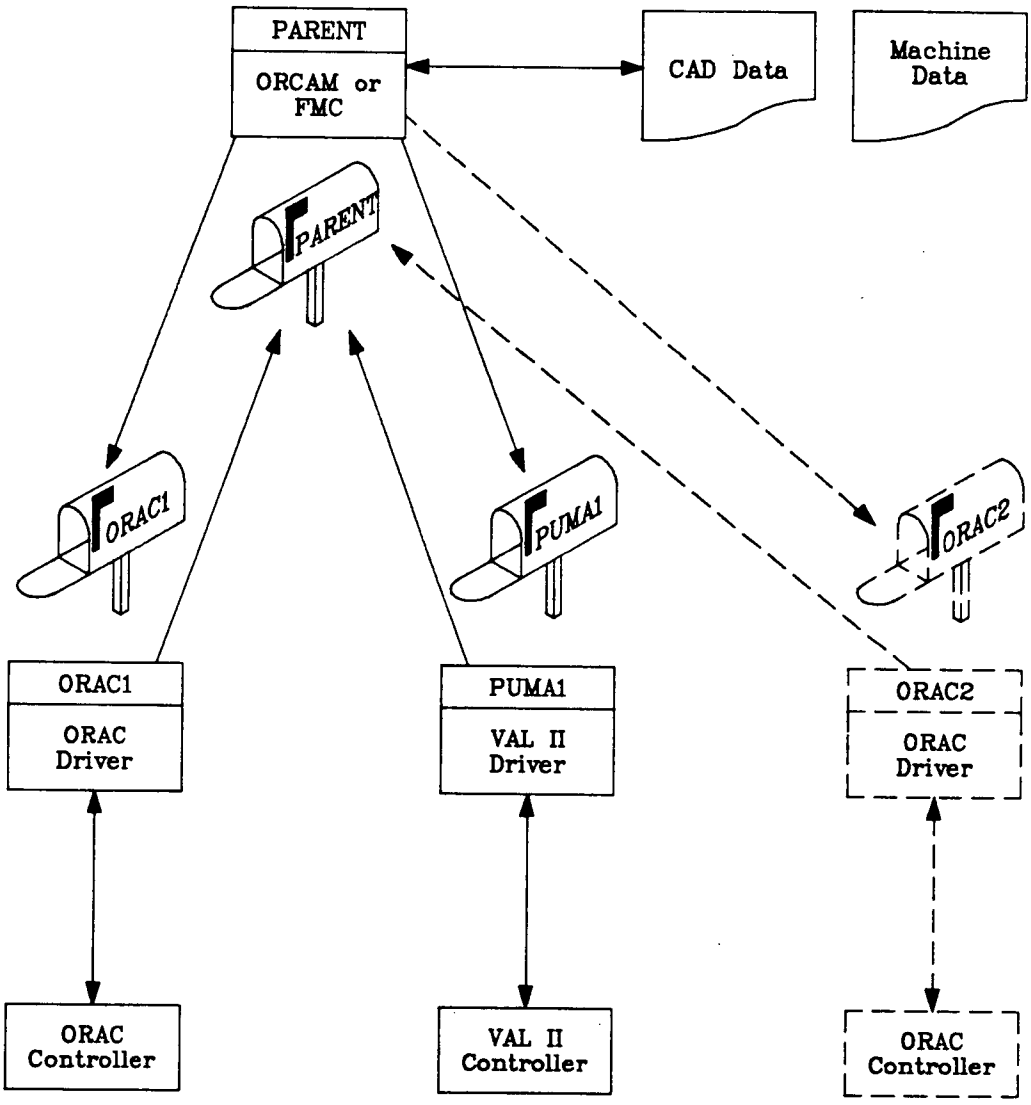

Figure 2.1: Task Distribution in the VAX/Machine Interface

Each machine driver subprocess is embodied with the particular characteristics of the machine and handles all of the direct communications. It is responsible for establishing communication with the machine controller, guiding the operator through the initialization sequence, overseeing the execution of commands received from the parent process, and maintaining a constant record of the machine's state. Because each machine has unique capabilities, each subprocess is also responsible for the detection and handling of errors. The corresponding error message and available recovery options are reported to the operator directly by the subprocess. The cell management software is informed of the error condition only if the recovery attempts fail to correct it.

2.2.1 Communication Channels

As shown in Figure 2.1, the different layers of the supervisory control system communicate in different ways. Communications between the middle layer and the machine control layer take place over physical paths, with at least one RS–232C serial line being used for each machine. The middle and top layers both reside on the VAX 11/750 so communications between these layers take place over internal data paths. These paths are realized by the creation of logical I/O devices known as *mailboxes*. This method of interprocess communication was chosen for the following reasons:

- The structure of each message is user defined so that all types of data may be transmitted, either as binary numbers or as ASCII coded character strings.
- The same I/O routines that are used for serial lines may be used for mailboxes by specifying the correct logical unit number.
- Messages may be read on either an interrupt or polled basis.

- The size of each mailbox may be individually assigned based on the maximum number of messages it will have to buffer.

When a process reads a mailbox message, the source of the message may be obtained directly from the VAX I/O routines. This allows a number of processes to write to the same mailbox without requiring an identifier in the actual message. The parent process takes advantage of this by using one mailbox to receive status messages from all of the subprocesses. Commands are transmitted to the individual driver subprocesses by placing them in the appropriate mailbox.

One additional mailbox that is not shown in Figure 2.1 is the *termination* mailbox. This is a special mailbox which the VMS operating system uses to notify the parent process if any of its created subprocesses terminate unexpectedly. This termination mailbox, and all other communication mailboxes, must be created and assigned a name by the parent process prior to the creation of the subprocesses. The termination mailbox name is then passed to the VMS system service as each new subprocess is created.

As illustrated in Figure 2.1, the communication mailboxes are given the same name as the associated subprocesses. These mailbox names are specified as the default input and output devices when the subprocess is created. This action causes the VMS system service routine to equate these device names to the process specific logical names SYS$INPUT and SYS$OUTPUT. These logical names allow the subprocess software to be written independent of actual mailbox names.

This independence is extended to the physical channels used to communicate with the machine controllers by setting up an additional logical name which points to the appropriate VAX terminal line. This logical name is created by the parent process and,

unlike those generated by the VMS system services, it is not process specific. Consequently, a unique name must be used for each subprocess. This uniqueness is generated by concatenating the characters "_TTY" to the subprocess name. This format makes it easy for each subprocess to access the correct I/O channel.

2.2.2 Status Messages

Table 2.1 lists the syntax of the four status messages that a machine driver subprocess may transmit to the parent process. The Startup message indicates that the subprocess is starting the initialization sequence and is the first message to be transmitted. The Ready message is transmitted after the initialization sequence is completed and indicates that the subprocess is ready to accept commands from the parent process. When a command is received it is translated and the corresponding machine actions are initiated. If all of the requested actions are successful, the Completed message is sent to the parent process. If an irrecoverable error is detected, the Error message is sent to the parent process along with the corresponding error number. As noted earlier, the syntax of the commands transmitted from the parent process to the subprocesses are machine dependant, and will be described separately for each machine driver.

2.3 Design of Operator Interface

The information that must be transmitted to the operator can be broken down into three categories; status, informational messages, and prompts. The status information comes primarily from the individual subprocesses and reflects the current state of both the machine and its controller. These state variables can generally take on a limited

Table 2.1: Machine Driver Status Message Syntax

Startup
Ready
Completed
Error <error number>

number of values and are well suited to a *status line* format in which each variable is assigned a particular position on the screen. This format allows the operator to scan the information rapidly at his own discretion, ensures that the most recent data is displayed, and prevents the display device from becoming cluttered with repetitive messages.

Informational messages and prompts are transmitted by both the parent process and the individual subprocesses. These messages are of variable length and content, and are best suited to a *dialogue* format rather than a menu structure. Both of these data types generally require the immediate attention of the operator and should therefore not be overwritten. One way of obtaining this protection is to use separate display windows for each source. Recent studies into attention theory have shown that this approach often results in both delayed and incorrect responses [4]. Ideally, all operator prompts should be displayed in one location and should be queued so that they may be serviced one at a time.

In order to handle these varying data types, it was decided to design the operator interface using an individual status window for each machine, and a combined message/prompt window to be used by all processes. These windows are generated using the Screen Management (SMG) services provided by VMS and are therefore independent of the actual terminal used. The SMG routines treat each window as if it were a separate virtual I/O device and assign it a unique identity number when it is created. All operations on the window are then performed by referencing this identity number. Once a window is created the SMG routines may be used to map the window to any portion of the actual output device, but it is up to the application software to ensure that previously *pasted* windows are not overwritten.

In a multitasking environment this window tracking is best performed by a separate task that implements all of the actual screen updates based on requests received from the other processes. In this way individual tasks do not have to be aware of each other's existence, and can simply request that a window be created in the next available location. This method was adopted for the supervisory control package, with the screen control task being created as a subprocess by the parent cell management process.

When deciding how to interface the screen management subprocess to the existing software structure, the following design criteria were used:

- The status windows associated with the individual machines should remain accessible for writing at all times.
- Each process should be able to gain exclusive control of the window used for messages and prompts. This type of access is necessary to prevent unanswered prompts from being overwritten, and to allow multiple line messages to be transmitted without interference from transmissions by other processes.

The VMS multitasking facility accommodates resource sharing through the Lock Management routines. These routines allow each process to request access to any supported I/O device in a number of different modes. If the request cannot be granted immediately, the process is placed in a queue until the device is available. Unfortunately, the lock management routines do not recognize the virtual windows created by the SMG services. A lock request to a display device therefore results in the entire display being dedicated to the requesting process.

To provide the desired separation of status and informational messages, the operator interface uses two different mailboxes as shown in Figure 2.2. Both of these mailboxes are created by the screen management subprocess and always have the same names. The SMG_hdplx mailbox is used to receive both status and control messages from the cooperating processes. It is set up as a unidirectional channel and is never written to by the SMG process. This mailbox is to remain available at all times and should not be used as the target of a lock request. The SMG_fdplx mailbox is used to receive all informational messages and prompts from both the parent process and the machine drivers. This mailbox is used as a bidirectional channel with all of the the operator's responses being passed back to the requesting process through the same mailbox. Each process should use the lock management routines to gain exclusive access to this mailbox before sending any messages.

The software necessary for performing the screen management utilities is located in the executable image SMG.exe. This software has been written as a general purpose operator interface and may be used for a variety of purposes. It operates on a 12 keyword command set which allows cooperating processes to create and write to a number of different windows. Each process may create up to four windows which are number

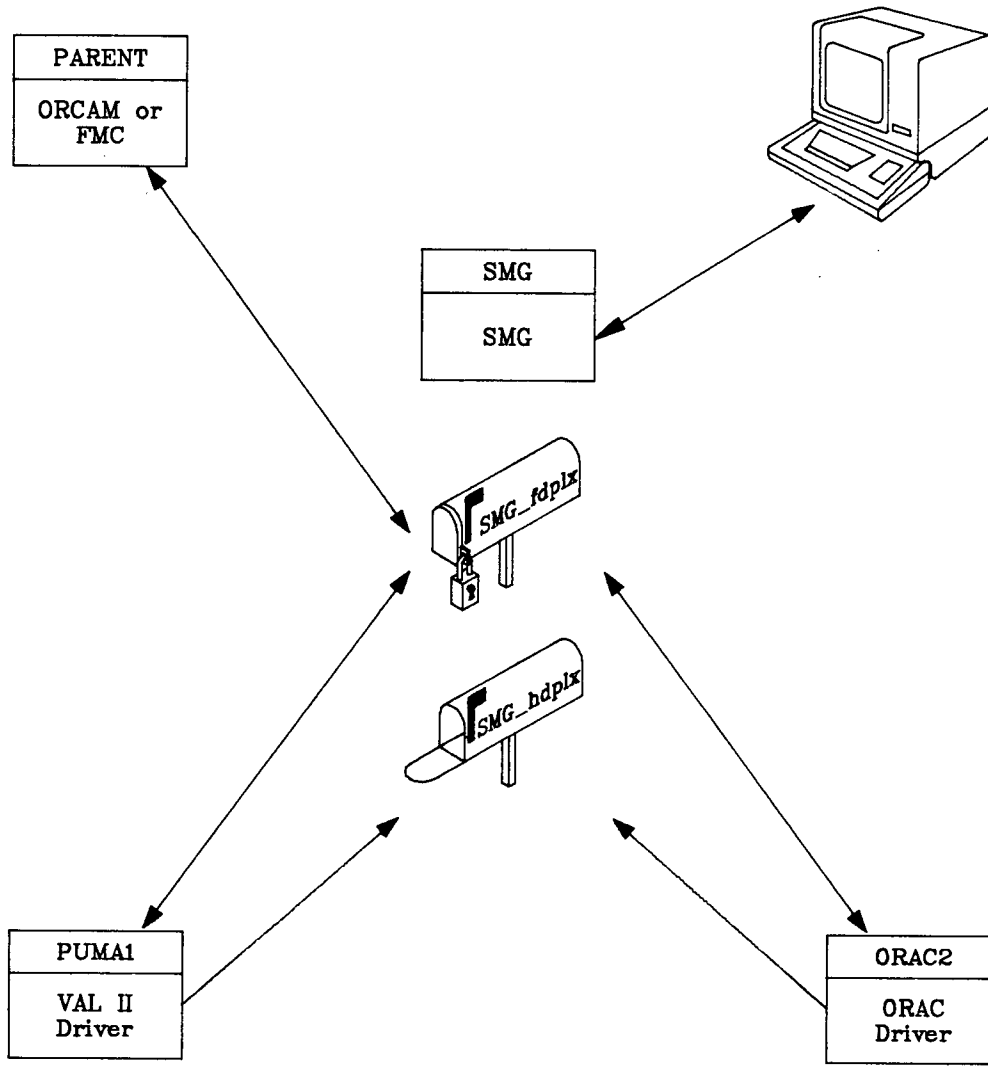

Figure 2.2: Communications with the Operator Interface

sequentially from 1–4. These windows are *owned* by the creating process and may not be written to by other processes. These windows may be used *as is* for displaying lines of text, or they may be subdivided into as many as 15 subwindows for displaying status information. In addition to these private windows, the SMG process maintains a single shared window which may be read from or written to by any of the cooperating processes. This window is referenced as window 0 and may not be subdivided by any process. When a process writes to this window the SMG software automatically appends the first 5 characters of the process name to the message before displaying it. This allows the operator to easily identify the source of the message or prompt.

The syntax of the 12 commands recognized by the SMG process is shown in Table 2.2. Keywords are shown in capital letters while variable parameters are indicated by a description of the required value enclosed in angle brackets. A choice between a number of valid keywords is indicated by a list of the keywords separated by vertical bars and enclosed in braces.

All processes must create their private windows by issuing the CREATE command. This command defines the size of the window and assigns a process specific identity number. The window will be enclosed by a border which may contain an optional label centered on either of the four sides. Once a window is created it may be subdivided using the CREATE_SUB command. Subwindows may only span one row, therefore its size is completely defined by the length parameter. This subwindow is assigned an additional identity number and may be placed anywhere within the main window by specifying the starting row and column. If the optional label parameter is supplied, the text is displayed to the left of the specifed window location. This label is only written once, when the window is created, and is not protected from overwriting by other windows.

Table 2.2: Screen Management Command Syntax

```
CREATE      <window> <rows> <columns> {TOP|BOTTOM|LEFT|RIGHT} <label>
CREATE_SUB  <window> <subwindow> <row> <column> <length> <label>

PUSH        <window> {UP|DOWN} <column> <space>
PASTE       <window> <row> <column>

UNPASTE     <window>
REMOVE      <window>

WRITE_SUB   <window> <subwindow> <text>
WRITE       <window> {NORMAL|REVERSE} <advance> <text>

RING_BELL   <window>
CLEAR       <window>
BATCH       <window> {ON|OFF}

READ  <window> <timeout> <prompt text>
```

Created windows do not become visible until they are pasted to the screen by either the PUSH or PASTE command. The PUSH command allows a window to be placed in the next available location starting at either the top or bottom of the screen. The column parameter may be used to specify the starting column for windows that are less than 80 characters wide. The space parameter determines how many blank lines will be left between the pasted window and its nearest neighbour, and defaults to zero if omitted. The PASTE command allows a process to specify an actual screen location for a window. This command forgoes the overwrite protection offered by the SMG process and should be used with caution.

The UNPASTE command is used to make a pasted window invisible. This command has the same effect regardless of which command was used to paste the window. Unpasted windows may still be written to and will reflect all updates when pasted back to the screen. The REMOVE command deletes the definition of the specified window from memory and makes it unaccessible. If the window is currently pasted to the screen it is automatically removed prior to deletion.

The WRITE_SUB command is used to write a status string to the specified subwindow. Both the window and subwindow must have already been created. The shared window is not subdivided and can't be accessed by this command. The remaining five commands are used to change the content of informational windows and may specify either a previously created private window or the shared window supplied by SMG. The RING_BELL and CLEAR commands are self explanatory. The WRITE command may be used to display a line of text in either normal or reverse video. The Advance parameter determines how many lines the cursor will advance after the message is printed, and may be set to zero to leave the cursor at the end of the message. The BATCH command may be used to group a number of messages together and have them displayed all at once rather than one at a time.

The READ command is used to send a prompt to the operator. The prompt text is displayed in the specified window, the terminal bell is rung, and the cursor is positioned at the end of the message. If the operator does not respond to the prompt before the Timeout period, the terminal bell is sounded again. All characters entered by the operator, up to and including the carriage return, are then written to the SMG_fdplx mailbox. As noted above, it is up to the requesting process to gain access to this mailbox prior to sending the READ command.

Chapter 3

ORAC LATHE INTERFACE

3.1 ORAC CNC Lathe

The ORAC lathe is a small 2-axis microprocessor controlled NC training lathe suitable for turning aluminum and other soft metals. The 0.37 KW AC main drive provides continuously variable spindle speeds up to 2000 rpm. The 20 mm bore spindle comes fitted with a manually operated, 3-jaw chuck. The standard self-centering, external grip chuck jaws can accommodate bar stock up to 40 mm in diameter. The longitudinal and cross slides ride on ballbearing leadscrews that have maximum travels of 350 mm and 95 mm, respectively. Each leadscrew has a pitch of 2 mm and is driven by a 200 step/revolution DC stepper motor; providing a single-axis resolution of 0.01 mm.

The ORAC CNC microprocessor is capable of performing both linear and circular interpolation. The velocity profiles required to trace the contour are sent to the corresponding drive units as two variable frequency pulse trains. The drive units amplify these TTL level control signals and drive the stepper motors in open-loop control, with no position or velocity feedback. The inaccuracies inherent in open-loop control make it an unusual choice for an NC contouring machine, but it has proven sufficient for the low accuracy work required of training systems such as the ORAC lathe.

Unlike the early DNC systems, the complete sequence of motions required to machine a particular part must be entered into the ORAC lathe's system memory prior to execution. This sequence must be coded in the native ORAC programming language; an operation *normally* performed via the NC function keys located on the front panel. In contrast to the cryptic codes utilized by early NC machines, the ORAC programming system provides a structured english-like language consisting of 17 instructions, including DO loops and subroutine CALLs. (A complete description of the programming language may be found in the ORAC Programming Instruction and Maintenance Manual [8].) Each instruction is stored in system memory as a separate *page*, which may contain up to nine lines of information. Each page occupies one of the 99 blocks of available memory. The lack of a file management system, however, limits the ORAC system to one part program at a time, regardless of the number of pages it uses.

To facilitate repeat operations, ORAC NC programs may be stored and retrieved from either the built-in minicassette recorder, or an external computer via an RS–232C serial communications line. Programs transmitted over the serial line are coded as ASCII text files with standard control characters to delimit the program, the individual pages, and the parameters within a page [13]. The serial interface is configured as Data Terminal Equipment and supports the DSR/DTR hardware handshaking lines. Software handshaking is not supported. The baud rate for serial transmission may be selected from the keypad to be either 300, 1200, or 2400 baud.

The ORAC programming language utilizes a standard X–Z coordinate system to define tool positions. The positive sense of these two axes is rigidly defined to the directions shown in Figure 3.1. The origin, however, is *floating* and may be set anywhere in the X–Z plane. Common practice is to set the origin at the intersection of the spindle

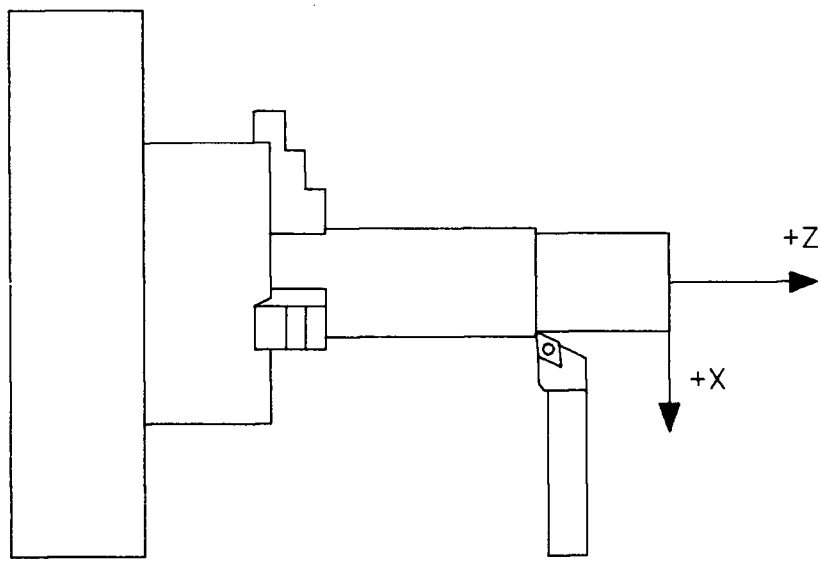

Figure 3.1: Coordinate Axis of ORAC Lathe

axis and the unclamped end of each new workpiece. This allows programs to be written independent of the actual clamping position. The coordinates for a given tool location may be specified either with respect to the floating origin—absolute mode, or with respect to the previous tool position—incremental mode.

The floating zero utilized on the ORAC lathe is not merely for programming convenience. The lack of any feedback devices (absolute or incremental) on the X and Z axis leadscrews means that the ORAC microprocessor has no way of determining the position of the toolpost with respect to the fixed spindle head. Each time the lathe is turned on, or a new NC part program is loaded, the microprocessor must be told where the program origin is with respect to the current slide positions. The Z axis zero point is set by facing the end of the workpiece with a reference tool, and then setting the internal Z axis counter to zero. The X axis zero is set by entering an accurate measurement of

a turned diameter. The internal X axis counter is then set equal to the corresponding radius.

The above procedure sets the floating zero with respect to a single tool. Depending on the geometry of a given part, more than one tool may be required to complete the machining operation. Each motion command in the ORAC programming language accepts a number between 1–9 to indicate which tool is to be used. (Tool number 0 is reserved for the reference tool used to set the floating zero point.) In order to produce an accurate part, the CNC microprocessor must know the location of the cutting edge of each tool. The geometries of tools 1–9 are recorded as offsets from the cutting edge of the reference tool, in both the X and Z directions. Once the offsets are recorded into memory, a new part program may be executed by simply resetting the floating zero point with tool 0. The current offsets may also be numerically edited to account for known amounts of tool wear. As with ORAC programs, the tool offsets may be stored on the minicassette unit for later retrieval. Unlike programs, however, the tool offsets cannot be transmitted over the RS–232C interface to an external computer.

The standard ORAC lathe comes equipped with a quick-change toolpost and holder capable of holding only one tool at a time. Whenever the NC program calls for a new tool, the microprocessor stops the spindle and the X and Z axes at their current position. The new tool number is displayed on the screen, indicated by a flashing cursor. The operator must then replace the current tool holder with the appropriate tool. Program execution is resumed by pressing the spindle start key followed by the axis start key.

The ORAC hardware provides four program accessible digital outputs and four digital inputs. The four solid state relay outputs are labelled A to D, and may be individually opened or closed via the AUXO instruction. The four voltage sensing inputs are labelled

E to H. The AUXI instruction may be used to pause execution of the NC program until a specified combination of presence and absence of input signals is attained. However, these input signals cannot be explicitly read or used as interrupts.

3.2 Pneumatic Tool Changer

In order to facilitate automatic operation and repeatable tool placement, a pneumatic tool changer was designed and mounted by Mr. Robert Moore of the Mechanical Engineering Department. The tool changer incorporates a rotating tool turret capable of holding four tools, one on each side. As shown in Figures 3.2 and 3.3, the tool turret is activated by two pneumatic pistons. The *clamp* piston raises or lowers the turret to disengage or engage the eight locking pins which mate the turret with the base. When in the lowered position, the turret is rigidly clamped to the base and may be used for machining. In the raised position, the turret may be rotated clockwise 45 degrees at a time by extending and retracting the *rotate* piston. The position of each piston is controlled by a 1/8 inch four-way two-position valve activated by a 24 V DC spring return solenoid.

The tool changer uses five micro-switches to provide feedback on the current position of the tool turret and the two pistons. Holes have been drilled in the bottom of the tool turret in patterns which allow the first three micro-switches to provide a binary representation of the angular position of the tool turret when in the clamped position. The fourth micro-switch is always closed when the tool turret is clamped, and open when the turret is in the raised position. The fifth micro-switch is open whenever the *rotate* piston is fully retracted, and closed otherwise.

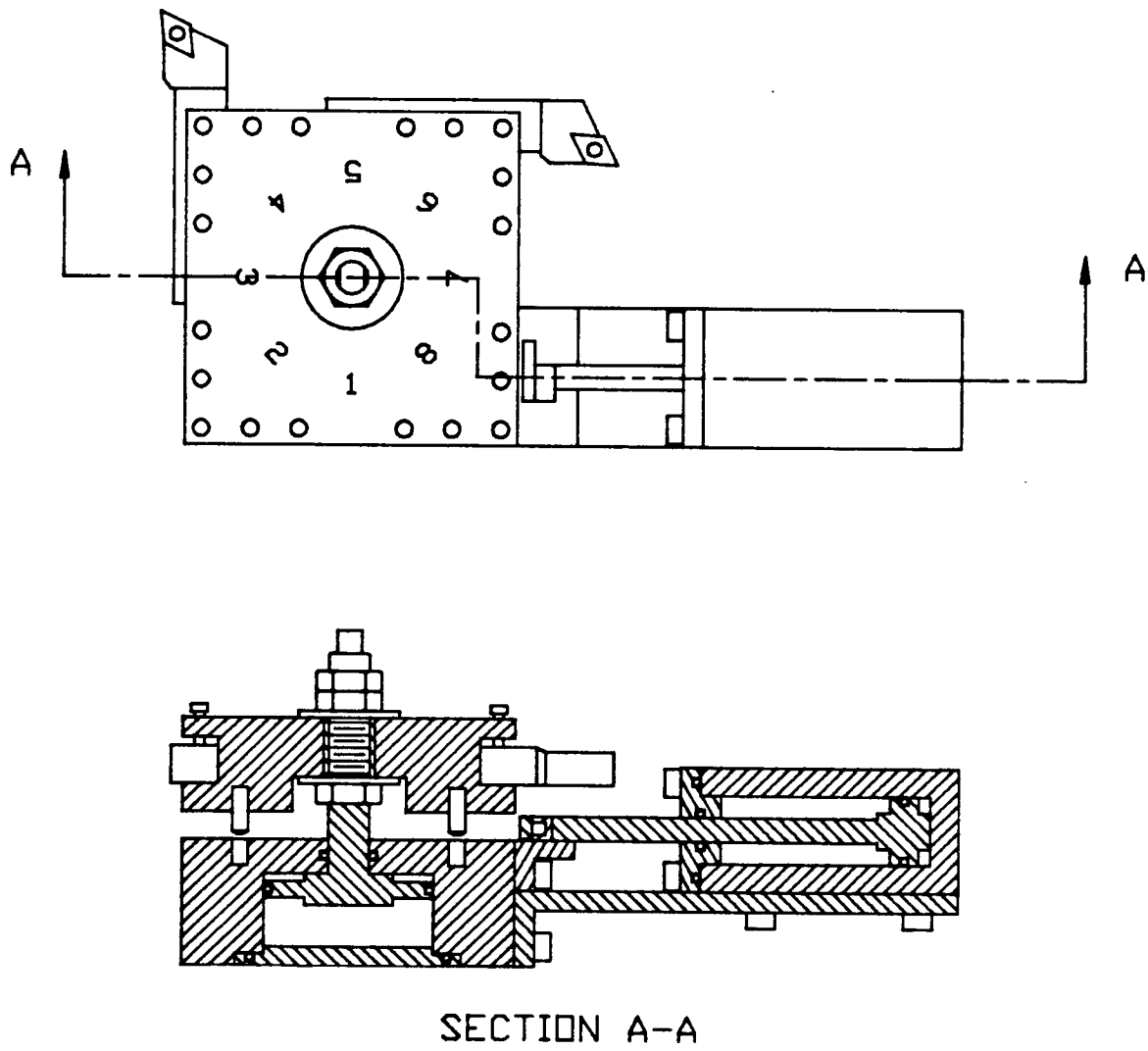

Figure 3.2: Pneumatic Tool Changer Assembly Drawing

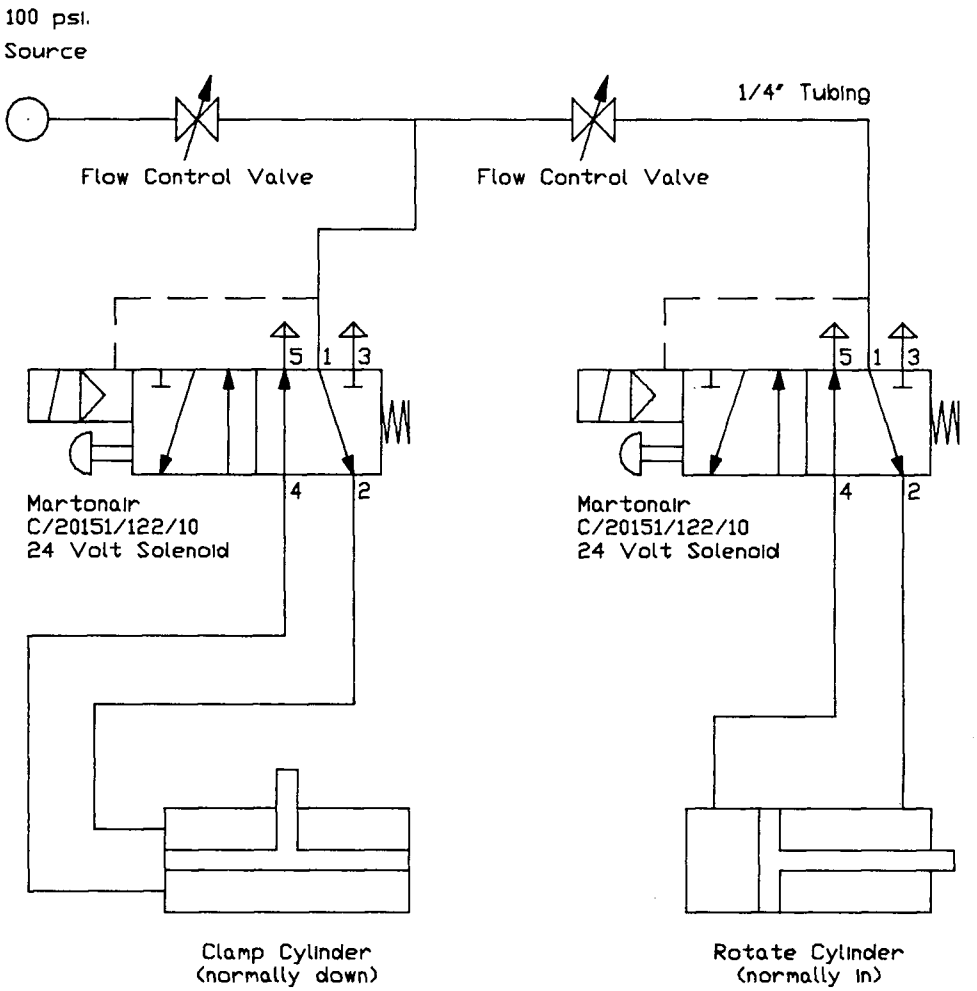

Figure 3.3: Tool Changer Pneumatics

The current turret position is also indicated by numbers stamped into the top face of the turret. These numbers range from 1 to 8 and indicate the current position by the number closest to the operator. Figure 3.2 shows the turret in position 1. Although the turret may be clamped in all eight positions, only the odd numbered positions provide a tool angle suitable for machining. The ORACAP off-line programming package for the ORAC lathe assumes that the tool turret will contain the following tools [13]:

Position 1 right-hand turning/facing tool for finish cuts
(also used as the reference tool)

Position 3 left-hand turning/facing tool

Position 5 threading tool

Position 7 right-hand turning/facing tool for rough cuts

3.3 Factors Influencing Design of Interface

The ORAC lathe is typical of many CNC machine tools in that it has been developed as a stand-alone unit with a predominantly manual interface. It has not been designed with the upwards communication channels necessary for unmanned operation. The standard configuration limits the microprocessor to accepting commands from the manual keyboard only, and to displaying status information on the built-in CRT only. The use of the serial communication port is limited to a DNC Behind-Tape-Reader interface accepting programs only.

Grieve and Smith [9] have suggested that the minimum state variables necessary to completely define the state of a CNC machine tool are:

- cycle on/off
- spindle on/off
- slide hold on/off
- chuck closed/open

- guards closed/open
- component position correct/incorrect
- tool condition acceptable/not acceptable

If, as with the ORAC lathe, these are not provided by the CNC microprocessor, then an intermediate programmable interface is necessary.

The ORAC lathe is essentially a black box closed system, which made the acquisition of even the minimum machine state difficult. The spindle is equipped with a digital encoder which permits both its speed and on/off state to be sensed. The X and Z axes, however, are not equipped with any feedback devices. The position "sensing" for these axis is performed in software by ROM based circuits embedded on multi-layered printed circuit boards, with limited access points. The guards and standard chuck are not automated, and therefore are not equipped with any sensing elements.

The manufacturers of the ORAC lathe were unwilling to provide adequate documentation to facilitate deciphering of the major hardware and software components. The large investment of time and funds which would be required to manually decode and modify the circuitry to provide a complete machine state sensing and an expanded serial port interface was determined to be outside the scope of this research project.

It was therefore decided to implement a programmable hardware interface which would duplicate the standard ORAC keypad. This hardware would provide the VAX software with access to the lathe's standard menu-driven user interface. The current menu level, however, would have to be inferred from the keys previously pressed. The hardware interface would also be responsible for the control and sensing of the pneumatic tool

changer. Because the ORAC CNC microprocessor does not report the tool requirements or the program execution status over the BTR serial line, this information must be communicated to the hardware interface by the NC program. This communication is accomplished by setting and/or clearing the appropriate auxiliary digital outputs.

At the time this research was initiated there was only one commercially available automatic chuck that was small enough to fit the ORAC's spindle. This chuck utilized high pressure air to both activate the chuck jaws and to hold them in place while the chuck was spinning. The stroke of the pneumatic actuator used to do this was so small that the chuck required three different sets of jaws to cover the full range of diameters machineable on the ORAC lathe. The high price and inflexibility of this chuck made it unsuitable for use in the proposed FMC. It was therefore decided to proceed under the assumption that an automatic chucking device would be designed in-house as a separate project, and that the hardware would eventually have to control this device as well. (This device has since been designed and tested, but has not been implemented due to the lack of a proper sized motor.)

Two possibilities existed for interfacing the necessary hardware with the control software executing on the VAX 11/750 computer. In addition to the standard RS-232C serial I/O ports, the VAX 11/750 architecture provides a 32-bit TTL level parallel I/O port. Use of the parallel port to directly control the hardware had the advantages of requiring less intelligence at the hardware level and faster transmission rates. However, the relatively large distance between the two machines, about 130 feet, created the disadvantage of high cabling costs. This distance was also outside the acceptable range for TTL level voltages and would require the control signals to be boosted to a higher voltage level for transmission.

Use of the RS–232C serial port to communicate with the interface had the disadvantages of requiring a UART (Universal Asynchronous Receiver/Transmitter) and associated hardware, as well as slower transmission rates. However, this option would result in lower initial cabling costs, and would allow later expansion of the control features of the interface without requiring additional cabling between the two machines.

Another factor influencing the choice of communications was the desire to adhere to the hierarchical design of the overall software system. As noted in Chapter 2, the close-to-machine functions provided by the hardware interface should ideally operate in an autonomic manner. This autonomy could be provided in software by creating a separate subprocess on the VAX, or it could be provided by designing an intelligent, microprocessor based hardware interface. As noted by Grieve, if too many autonomous subsystems are implemented at higher levels of the hierarchy, there is a great risk of creating a control environment that is top-heavy and difficult to implement and debug [9]. It was therefore decided to delegate the majority of the monitoring and control functions to a microprocessor based interface which would act upon commands received from the VAX software over the same serial line as used for downloading ORAC programs.

After looking at several single-board microprocessor based control systems, it was decided to build a complete interface from scratch based on the ZILOG Z8671 MCU 8–bit single-chip microcomputer. This 40 pin self-contained computer provided the following attractive features:

- An onboard full duplex UART providing TTL level serial I/O communications which can be operated in either an interrupt or polling mode. One of two on-chip timer/counters provides the bit rate for eight different hardware selectable baud rates, thus minimizing the need for external circuitry.

- 2K of ROM preprogrammed with a BASIC/DEBUG interpreter complete with an interactive line editor and debugger for quick and easy program development. The interpreter also provides a complete interface to machine language routines for operations requiring fast execution or features not directly supported by BASIC/DEBUG.
- 32 I/O lines configurable under software control to provide timing signals, status signals, 4 prioritized maskable hardware interrupts, parallel I/O with or without handshake control, and/or an expanded address/data bus for interfacing external program and data memory.
- Automatic execution of programs stored in external ROM on power-up or reset without entering the BASIC/DEBUG monitor.

3.4 Hardware Interface

3.4.1 Z8671 Microcomputer

The 40 external access pins of the Z8671 MCU are comprised of 32 I/O lines, 6 timing and control lines, and 2 power supply lines. The function and pin assignment for each of these lines is shown in Figure 3.4. A functional block diagram showing the relationship between these external lines and the internal architecture of the Z8671 is provided in Figure 3.5 [18].

The Z8671 utilizes a 16–bit program counter to access three different address spaces; program memory, data memory, and a register file. Program memory consists of 2K bytes of internal ROM and up to 62K bytes of external ROM, EPROM, or RAM located between addresses 2048 and 65535. The first 12 bytes of internal program memory

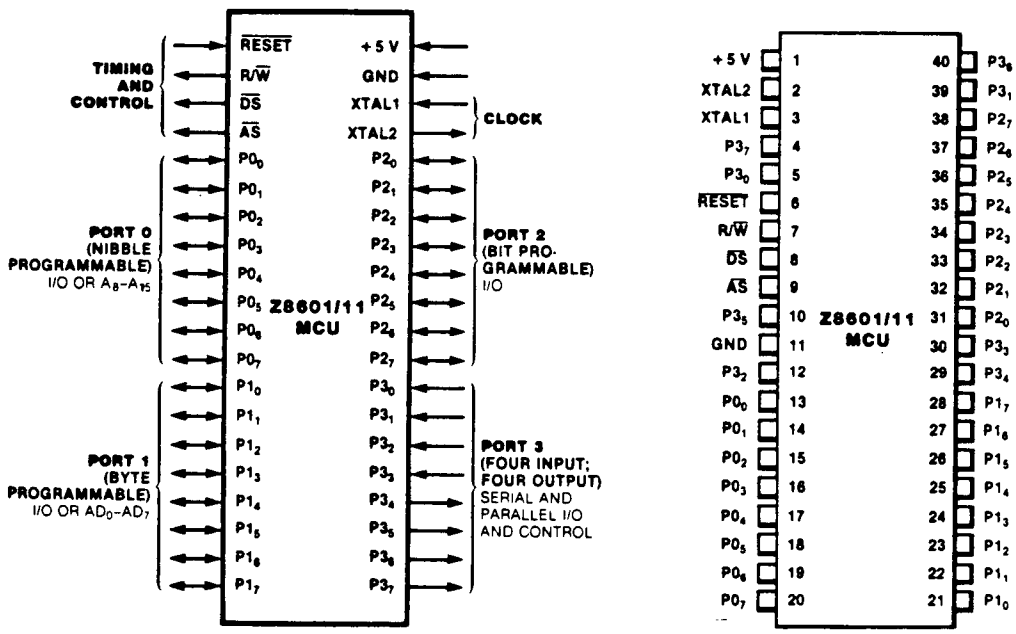

Figure 3.4: Pin Functions and Assignments for Z8671 MCU

are reserved for six 16–bit interrupt vectors. The BASIC/DEBUG interpreter, which occupies the remaining 2036 bytes of internal ROM, sets these interrupt vectors to point to addresses 1000–1011 hex of external memory.

Data memory consists of up to 62K bytes of external ROM, EPROM, or RAM also accessed between addresses 2048 and 65535. In order to distinguish between data and program memory, one of the I/O lines can be programmed as a $\overline{\text{DM}}$ control line. When configured in this way, the Z8671 will pull the $\overline{\text{DM}}$ line to the active low state during all external memory references which are not *instruction fetch* cycles.

The 144–byte register file contains 4 I/O registers, 124 general purpose registers, and 16 status/control registers. BASIC/DEBUG maps these registers to program memory addresses 0–3, 4–127, and 240–255 respectively, so that they may be individually accessed

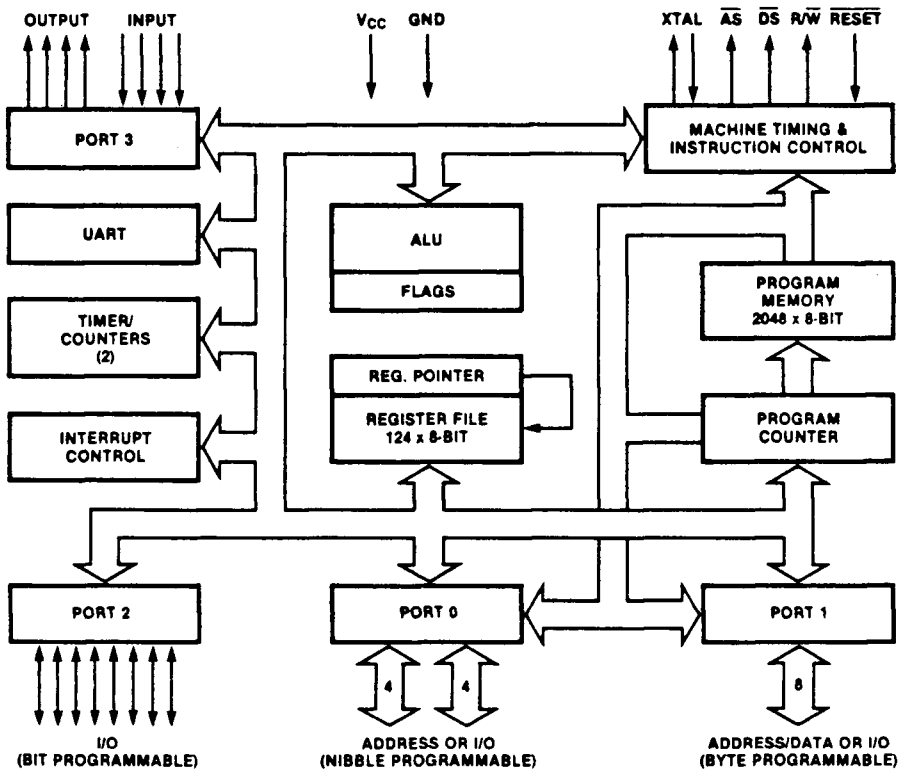

Figure 3.5: Functional Block Diagram of Z8671 MCU

by application programs. To perform I/O operations, the software simply reads or writes to the appropriate address, 0–3, the same as if an external memory location was being referenced. BASIC/DEBUG uses up to 94 of the general purpose registers as pointers, internal variables, and scratch pad memory, leaving only 30 registers completely free for use by the application software.

The Z8671 MCU's 32 I/O lines are grouped into four 8–bit ports which may be individually configured by the application software. Port 1 is byte programmable as either an input port, an output port, or a multiplexed address/data bus for accessing both external program memory and external data memory. When used as an I/O port, Port 1 may be placed under handshake control. In this configuration, Port 3 lines $P3_3$

and $P3_4$ serve as the RDY_1 and $\overline{DAV}_1$ control lines which synchronize the data transfer between the Z8671 and the external device. The RDY_1 signal is set by the device reading the port whenever it is ready to accept new data. The $\overline{DAV}_1$ signal is cleared by the output device whenever new data has been written to the port.

When used to access external memory, Port 1 provides both address lines A_0–A_7, and data lines D_0–D_7. The address lines are output first, and are valid for 70 ns after the trailing edge of the active low Address Strobe, $\overline{AS}$. During a write operation, the data provided by the Z8671 is valid on the leading edge of the active low Data Strobe, $\overline{DS}$. During a memory read operation, the Z8671 MCU latches the data on the Port 1 lines on the trailing edge of the $\overline{DS}$ pulse. Figure 3.6 gives a complete timing diagram for both the read and write cycles.

Port 0 is hardware configured as two 4–bit *nibbles*. Each nibble may be individually programmed as either input, output, or an expanded external address bus. When used as an I/O port, the upper nibble of Port 0 may be placed under handshake control using Port 3 lines $P3_2$ and $P3_5$ as the RDY_0 and $\overline{DAV}_0$ signals. When used for accessing external memory, the lower nibble provides address lines A_8–A_{11}, and the upper nibble provides address lines A_{12}–A_{15}.

The 8 I/O lines of Port 2 may be individually configured as either inputs or outputs. Like the other I/O ports, Port 2 may also be placed under handshake control utilizing Port 3 lines $P3_1$ and $P3_6$ as the RDY_2 and $\overline{DAV}_2$ signals. The direction of the handshake signals is determined by the I/O function assigned to bit 7 of Port 2, and only those bits which are assigned the same function should use the handshake signals.

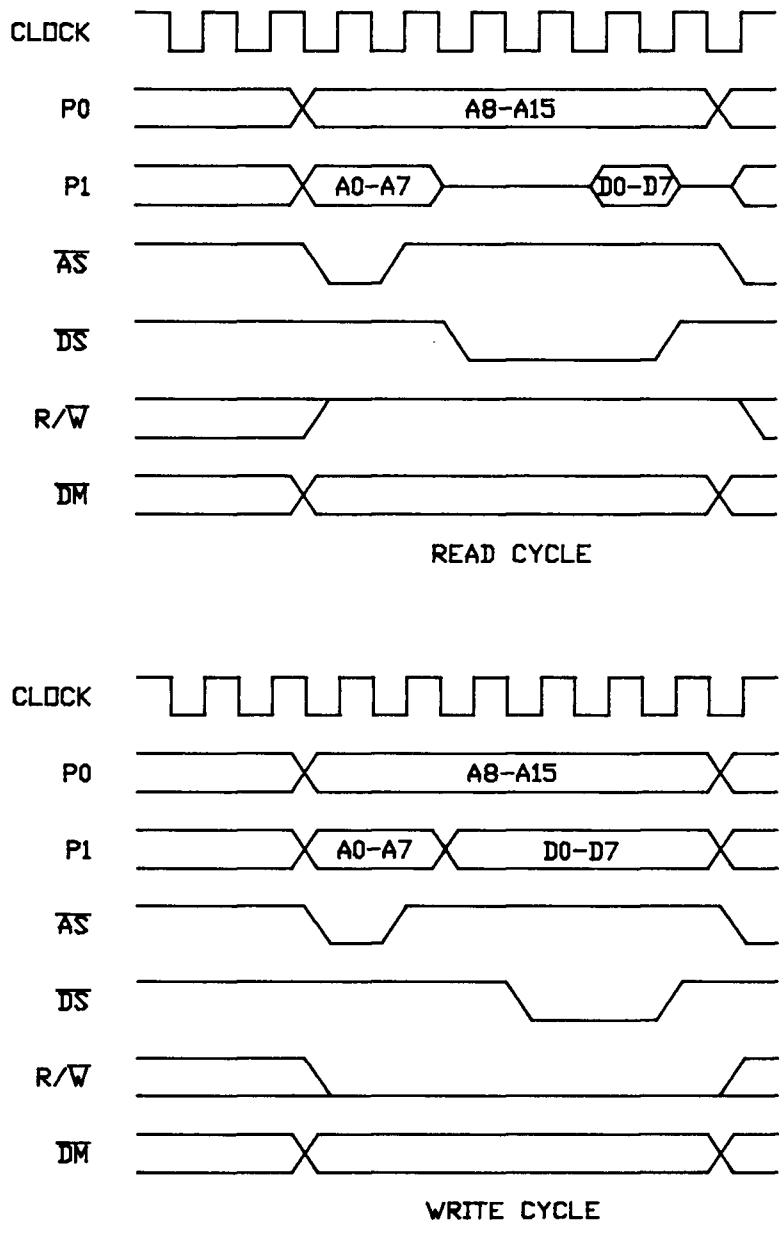

Figure 3.6: Timing Diagram for Z8671 READ and WRITE Cycles

Port 3 is hardwired as four input lines (low order nibble) and four output lines (high order nibble). In addition to the handshake functions already described, the four output lines of port 3 may be configured to provide serial output, internal timer/counter output pulses, and a $\overline{DM}$ control signal to reference external data memory as opposed to external program memory. The four input lines may be configured to accept serial input, external timer/clock input, and up to 4 external interrupt request signals.

The BASIC/DEBUG interpreter allows quick examination and modification of all external memory locations. It is possible, therefore, to expand the I/O capabilities of the basic Z8671 system by providing external I/O devices which are mapped into the external memory space. BASIC/DEBUG utilizes this technique by reading external memory address FFFD hex once upon start-up or reset to locate the user-selected baud rate for serial communications.

3.4.2 External Memory

The BASIC/DEBUG interpreter occupies the total 2K of internal memory available on the Z8671, therefore, application programs must be stored in external memory. The required amount of external program memory may be located anywhere in the available address space. On power-up or reset, BASIC/DEBUG searches for external RAM by performing a nondestructive test of the memory space from low to high addresses. Only one byte is tested for each 256–byte page at relative location xxFD hex. If RAM is found at the byte tested, the interpreter assumes that the entire page is RAM. Testing continues until a byte is found that does not contain RAM. The interpreter then checks for an automatic start-up program stored in ROM. The start-up program must begin with a BASIC line number between 1 and 254, and start at location 1020 hex.

When BASIC/DEBUG discovers RAM in the system, it allocates the top page of external RAM for the serial input line buffer, the subroutine stack, and the 26 named variables. If no RAM is discovered, BASIC/DEBUG maps these functions to the register file. The line buffer is multiplexed with the expression evaluation stack, and the subroutine stack is multiplexed with the variables [19]. This limits the number of variables and the length of expressions which may be used by the application software. In each case, however, the appropriate pointer registers are initialized to mark the boundaries between each area that is assigned a specific use.

The BASIC program which controls the ORAC lathe requires approximately 3K of memory and utilizes most of the 26 variables available in BASIC/DEBUG. During program development and testing it was more expedient to load the various sections of code into RAM rather than ROM. Completed sections could be permanently stored on the VAX 11/750 and downloaded when necessary. For final operation, however, it was felt that the program should be loaded into ROM so that the auto-start feature could be utilized. This would eliminate the need for continual downloading from the VAX each time the system was powered up or reset. The memory interface circuitry was therefore designed with two 24–pin sockets which may be configured for either a RAM or ROM system. For development purposes, both sockets are loaded with 2016 2K RAM's. For automatic start-up, a 2732 4K ROM and a 2016 2K RAM are used. The 2K of RAM in the automatic start-up system allows all 26 variables and complicated expressions to be used.

The complete external memory interface circuit is shown in Figure 3.7. I/O ports 0 and 1 are configured as memory reference ports providing the full 16–bit address bus to the external circuitry. The multiplexed address bits are latched using a TTL 74LS373

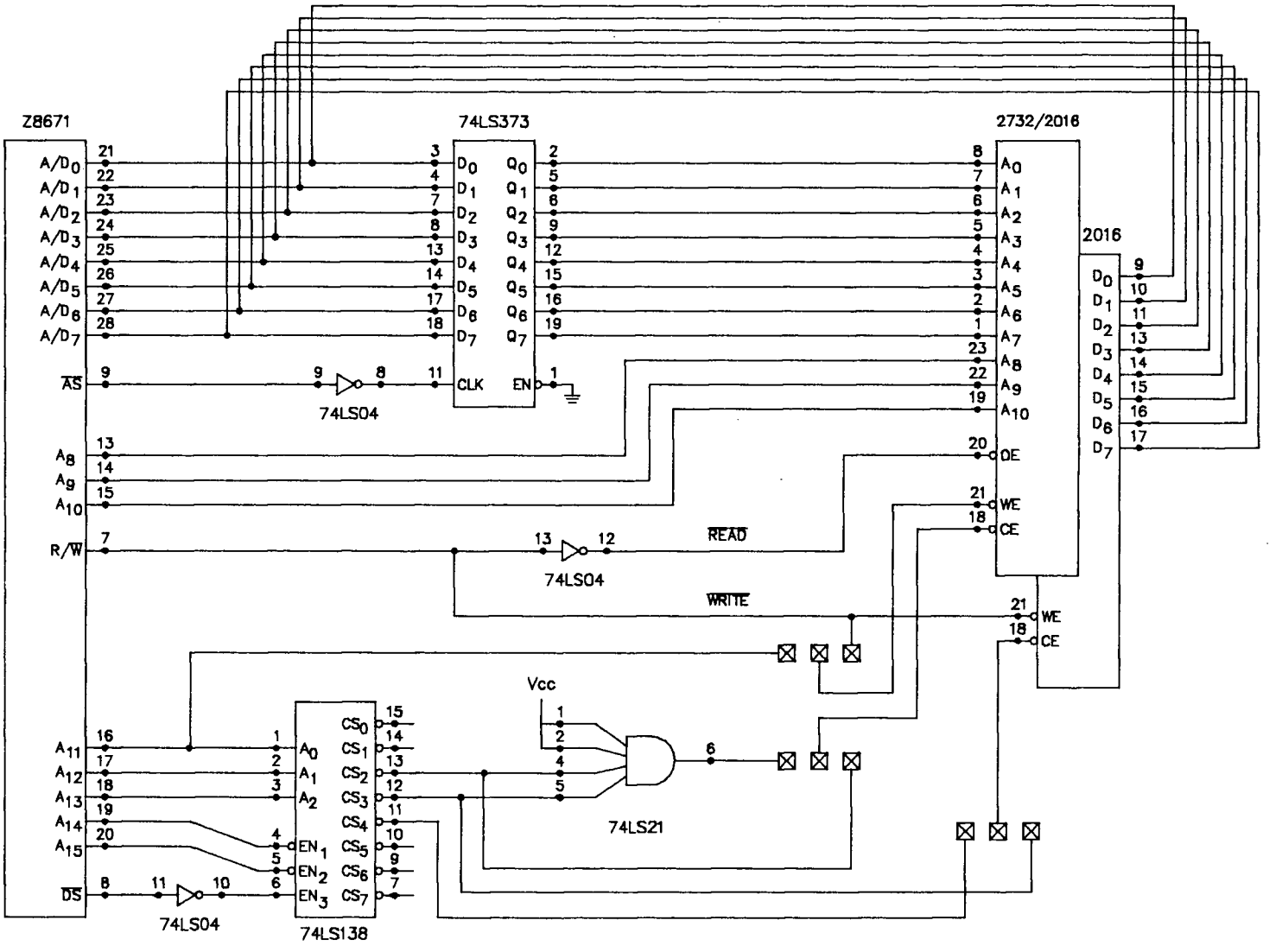

Figure 3.7: Address Decoding Circuit for External Memory

octal latch clocked by the inverted $\overline{AS}$ strobe. Address bits A_{11}–A_{15} are decoded using a TTL 74LS138 3–to–8 line decoder and a 74LS21 AND gate to provide the necessary memory select lines. Three 2K RAM select signals are provided at addresses 1000, 1800, and 2000 hex; as well as one 4K ROM select signal at address 1000 hex.

Two jumper pads are provided for setting the configuration of the left-hand memory socket. With both jumpers placed to the right, the socket is configured for a 2K RAM using address bits A_0–A_{10}, and both a $\overline{READ}$ and a $\overline{WRITE}$ strobe. With both jumpers placed to the left, the socket is configured for a 4K ROM using address bits A_0–A_{11} and a $\overline{READ}$ strobe only. In both cases the starting address is set to 1000 hex. This facilitates both automatic start-up of programs starting at 1020 hex, and the processing of interrupts, which are vectored to locations 1000–1011 hex.

The right-hand memory socket is always configured for a 2K RAM, but the starting address is jumper selectable. If the left-hand socket has been configured for a 4K ROM, the jumper should be placed to the left. This will map the 2K RAM directly below the 4K ROM at address 2000 hex. If the left-hand socket has been configured for a 2K RAM, the jumper should be placed to the right. This will map the socket to address 1800 hex, creating a contiguous 4K block of RAM.

3.4.3 Serial Input/Output

The serial I/O lines provided by the VAX 11/750 architecture meet the EIA RS–232C specifications. In order to interface these ± 12 V RS–232C signals to the TTL level signals required by the Z8671's onboard UART, the serial I/O interface circuit uses Motorola MC1488 line drivers and MC1489 line receivers. As shown in Figure 3.8, the output

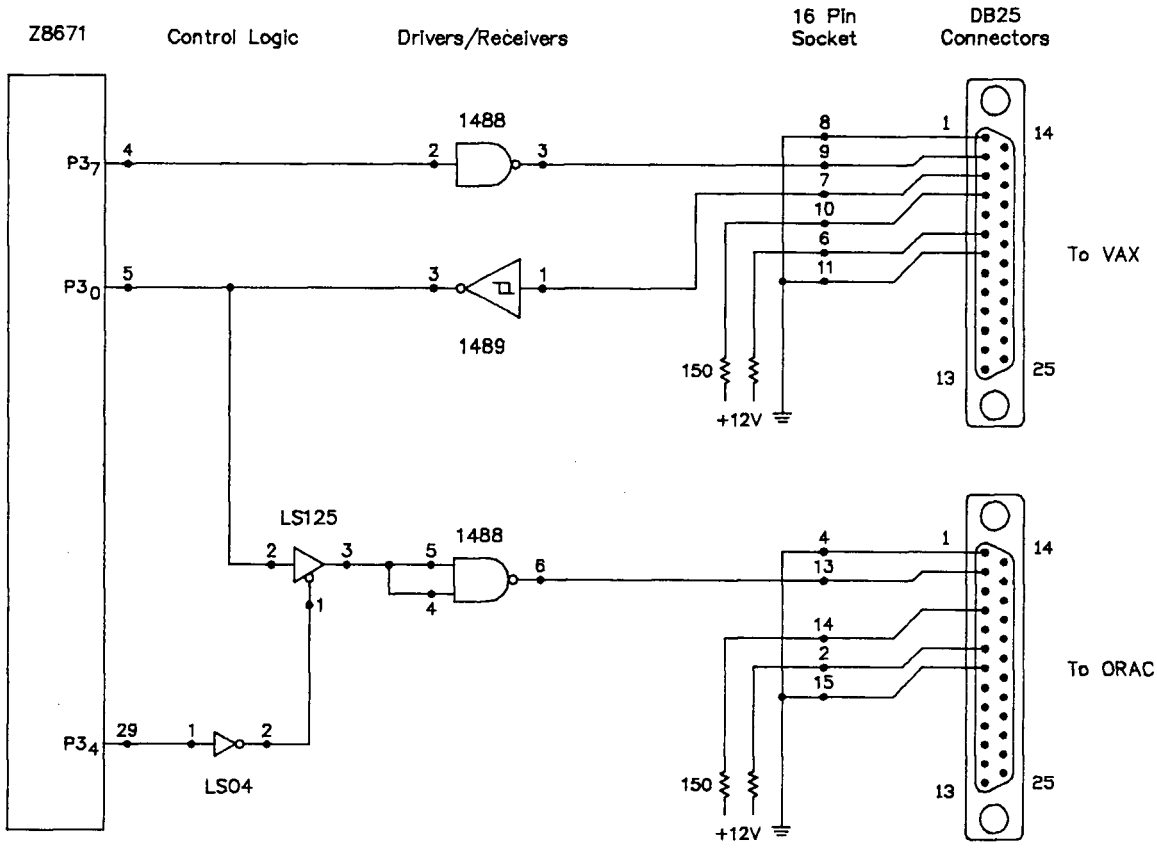

Figure 3.8: Z8671 Serial I/O Interface

from the RS–232C receiver is directed both to the Z8671's serial input pin, $P3_0$, and to an additional RS–232C driver connected to the ORAC lathe's BTR serial interface. This redirection takes place through a tri-state buffer which is gated by bit 4 of I/O Port 3, which is configured as an output. This arrangement permits the Z8671 software to selectively allow the ORAC lathe to *listen in* on information received from the VAX, such as when NC programs are being transferred from the VAX to the Z8671 interface. By multiplexing ORAC NC programs and Z8671 commands onto one serial line, the automatic echo feature of the Z8671 can be used to detect transmission errors on all communications. This is a feature not provided by the ORAC BTR interface.

3.4.4 Control Signals

I/O Port 2 has been configured as an 8–bit output port under handshake control to provide the control functions necessary to operate the interface. In order to expand the number of control functions available, these 8 output lines are externally divided into a 4–bit data bus and a 4–bit address bus. The upper nibble of Port 2 provides the address bus which is decoded into 16 function select lines. The lower nibble of Port 2 provides the data bus which is gated to the input lines of the circuitry providing each function. In the present design, only 4 function select lines are in use, leaving 12 functions for future expansion.

The circuitry used to decode the 4–bit address bus is detailed in Figure 3.9. The address present on pins $P2_4$–$P2_7$ is strobed into the 4–bit input latch of a CMOS 4515 4 to 16 line decoder on the leading edge of the $\overline{DAV}_2$ signal. The decoded function select line changes from high to low after a propagation delay of approximately 500 ns. The input circuit of the selected function uses this falling edge to latch the data present on the data bus, $P2_0$–$P2_3$.

In order to ensure that the Port 2 output remains valid long enough for the slowest function to latch the data, two monostable multivibrators are used in series to delay the RDY_2 input signal. The first monostable multivibrator is triggered on the leading edge of the $\overline{DAV}_2$ pulse, and is configured for non-retriggerable operation. The timing resistor and capacitor have been selected to provide a 33.3 ms active low output pulse. The trailing edge of this pulse triggers the second monostable multivibrator, producing a 57.5 ns active low pulse on the RDY_2 line. The Z8671 MCU responds to this high to low

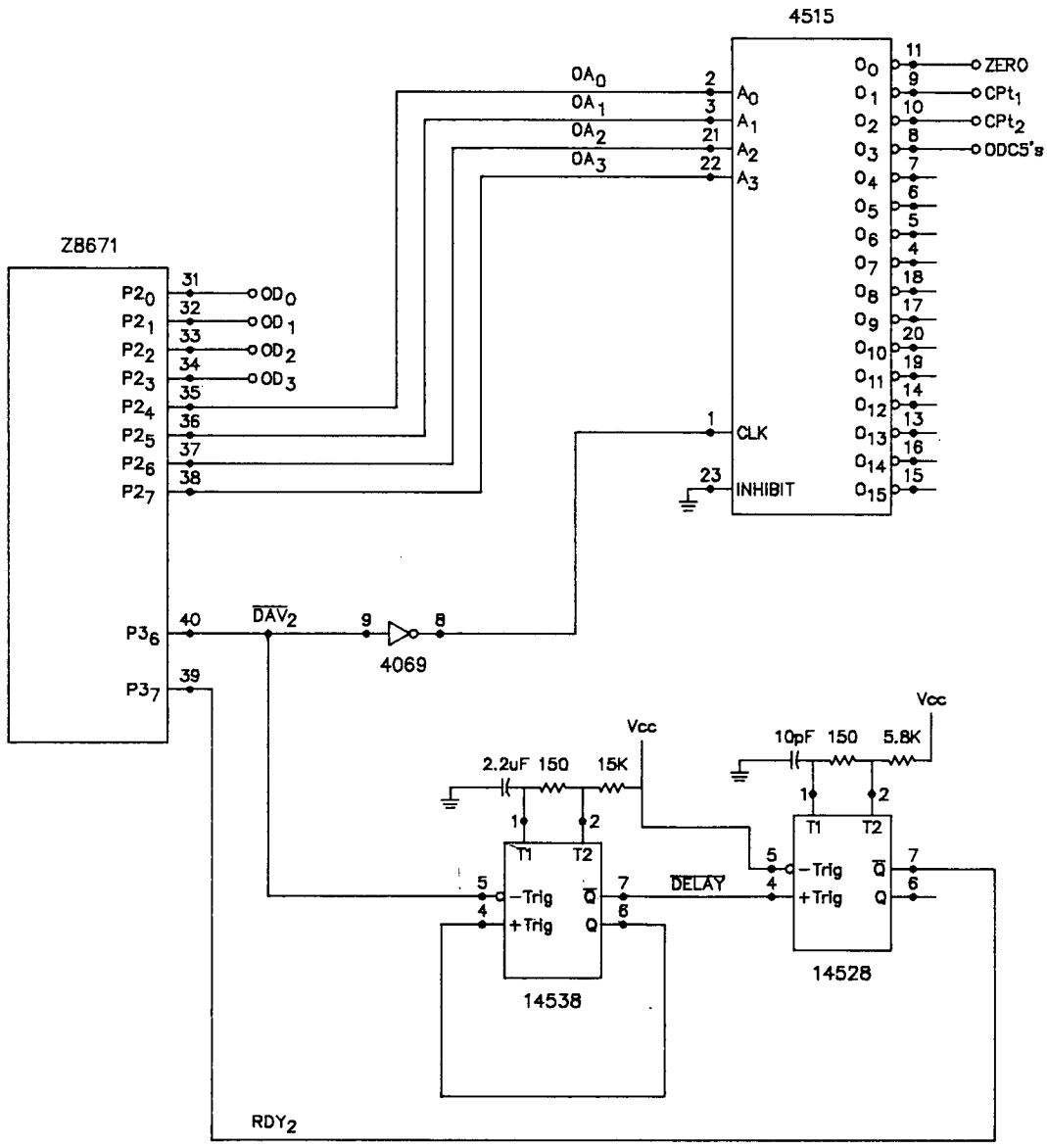

Figure 3.9: Decoding Circuit for Port 2 Address Bus

transition on the RDY_2 line by pulling the $\overline{DAV}_2$ line high, which completes the output operation.

The input latches of the 4515 decoder hold the function select line in the active low state after the Z8671 output operation has completed. In order to generate a function select pulse, such as required by an edge triggered latch, the required function must first be selected, then "unselected". Output address zero has been reserved for this function and is therefore not connected to any driver circuitry. Each output operation to a non-zero address must be followed by an output operation to address zero, which will reset the selected function driver.

Output addresses 1 and 2 are used to access two identical CMOS 22102 cross point array drivers. Each driver provides 2 arrays of 16 switches arranged in a 4x4 cross point pattern. Each of the 16 switches may be individually selected for opening or closing by presenting the appropriate 4–bit binary code to the on-chip decoder. Complementary switches in both arrays always have the same switch setting. Together, these two IC's provide the interface to the ORAC lathe's manual keypad.

The output signals from the 54 key ORAC keypad (Figure 3.10) are provided to the CNC microprocessor on a 34–pin ribbon cable. These 34 signals were decoded using a digital logic analyzer and map to the keypad functions as detailed in Table 3.1. The data entry and NC programming keys are arranged in a 5x8 cross point array, whereas the manual operation and spindle control keys are each individually switched to system ground.

The signal lines corresponding to the Finished (F), Enter (E), Axis Start, Axis Stop, Spindle Start, Spindle Stop, cursor positioning, and numeric keys have been connected

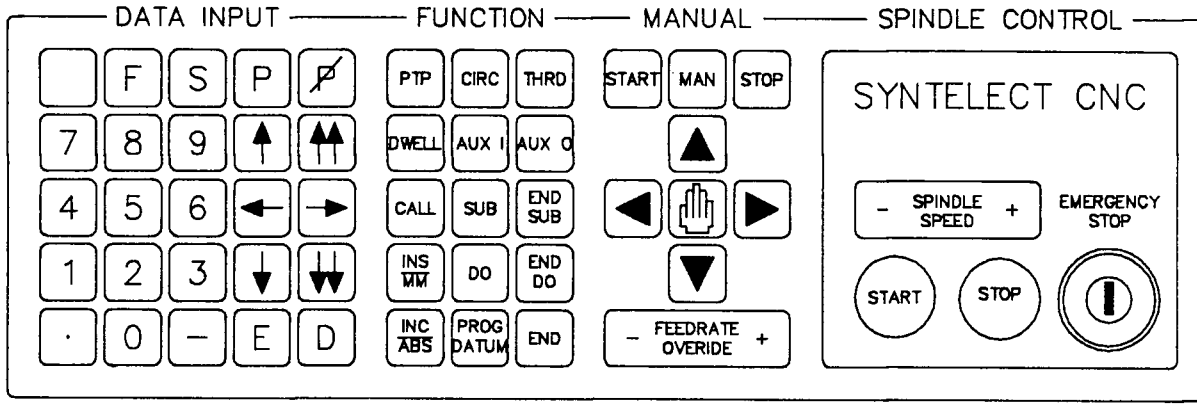

Figure 3.10: Layout of ORAC Keypad

Table 3.1: ORAC Keypad Signal Decoding

Pin #	Keypad Signal		Pin #	Keypad Signal	
	Section	Function		Section	Function
1	Data/Function:	Row_1	18	Manual:	Feedrate –
2		Row_2	19		Jog Left
3		Row_3	20		Jog Forward
4		Row_4	21		Jog Speed
5		Row_5	22		Jog Right
6		$Column_1$	23		Feedrate +
7		$Column_2$	24	Spindle:	Spindle Start
8		$Column_3$	25		Spindle Speed –
9		$Column_4$	26		Spindle Stop
10		$Column_5$	27		Spindle Speed +
11		$Column_6$	28	Other:	System GND
12		$Column_7$	29		System GND
13		$Column_8$	30		System GND
14	Manual:	Axis Stop	31		+ V Buzzer
15		Manual Mode	32		+ V Buzzer
16		Axis Start	33		– V Buzzer
17		Jog Backward	34		– V Buzzer

to the 4x4 arrays of the two 22102 IC's as shown in Figure 3.11. (The NC programming and manual jog keys are not necessary for automatic operations.) The control lines of the 22102 chips are wired such that the switch specified by the 4–bit data bus is turned on when the cross point array chip is selected, and turned off when the chip is "unselected". Table 3.2 lists the Port 2 output values necessary to access each of the available keys.

Output address 3 selects a CMOS 14175 quad D–type latch. As shown in Figure 3.12, the latched data is used to drive three optically isolated ODC5 output modules. A SN75468 octal Darlington transistor array is used to boost the current capacity of the CMOS outputs to the input requirements of the ODC5 modules. Two of the output modules are used to switch 24 volts to the two solenoid valves on the pneumatic tool changer. The third module is wired to digital input E on the ORAC lathe. The available output codes and their actions are listed in Table 3.2. Each ODC5 output module is also complemented by a manual toggle switch. The MANUAL/AUTO switch on the Z8671 cabinet determines whether the ODC5 modules or the toggle switches are active. These manual switches allow the tool changer to be operated when the system is not under computer control.

In addition to the control signals provided by Port 2, bit 5 of Port 3 is configured as an output signal to provide an emergency stop function. The inverted $P3_5$ output line drives an optically isolated OAC5 AC output module. The output side of the OAC5 module is wired in series with the emergency stop keyswitch on the ORAC lathe's front panel. This control function is implemented differently from the other control functions to provide a shorter reaction time, and to facilitate access from Z8 machine language routines as well as BASIC/DEBUG programs.

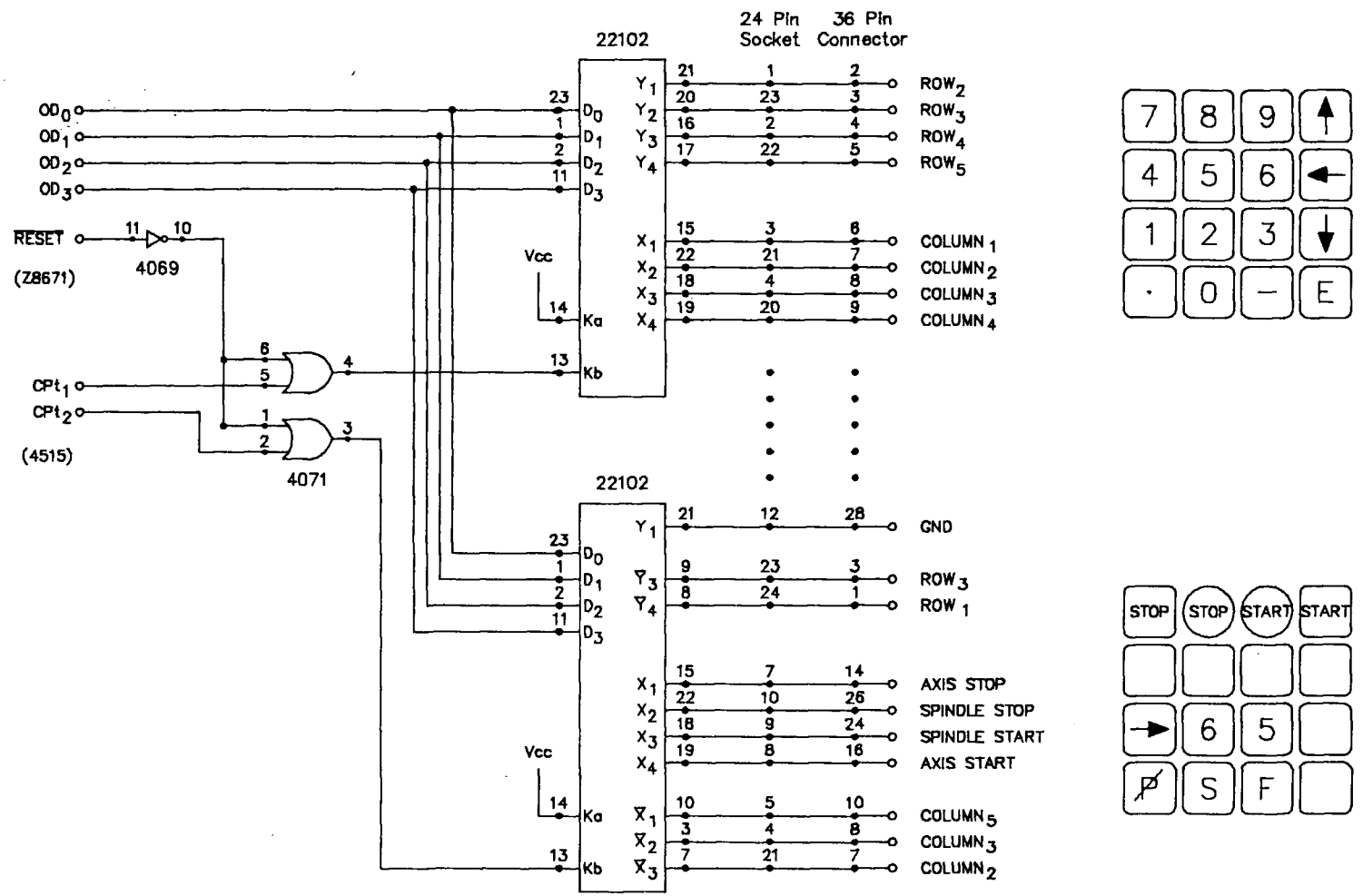

Figure 3.11: ORAC Keypad Interface Circuit

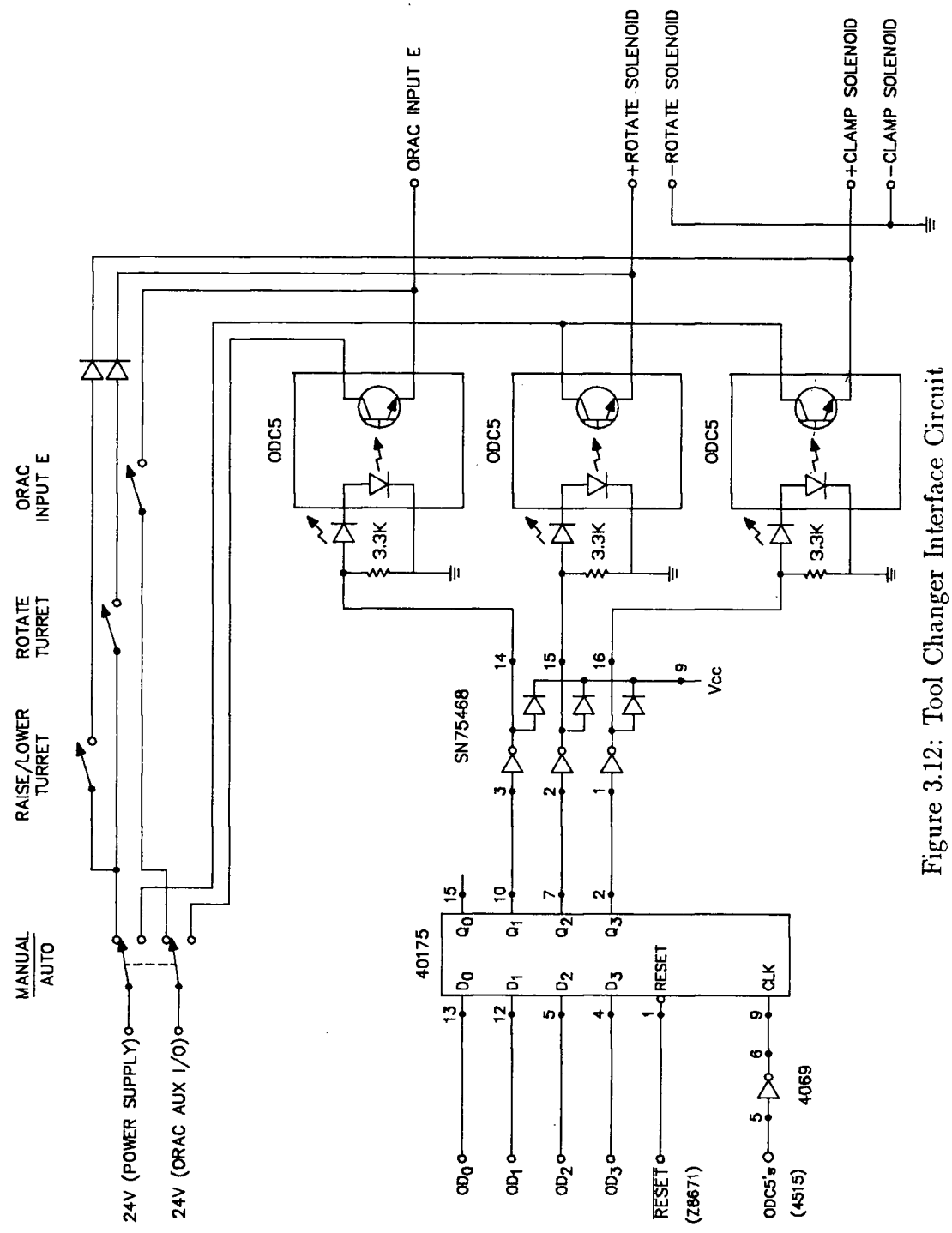

Figure 3.12: Tool Changer Interface Circuit

Table 3.2: Port 2 Output Codes and Functions

Output Code			Function	
Binary	Hex	Decimal	Device	Action
0000 xxxx	00–0F	0–15	All IC's:	Reset
0001 0000	10	16	ORAC Keypad:	7
0001 0001	11	17		8
0001 0010	12	18		9
0001 0011	13	19		Cursor Up
0001 0100	14	20		4
0001 0101	15	21		5
0001 0110	16	22		6
0001 0111	17	23		Cursor Left
0001 1000	18	24		1
0001 1001	19	25		2
0001 1010	1A	26		3
0001 1011	1B	27		Cursor Down
0001 1100	1C	28		.
0001 1101	1D	29		0
0001 1110	1E	30		–
0001 1111	1F	31		E
0010 0000	20	32	ORAC Keypad:	Axis Stop
0010 0001	21	33		Spindle Stop
0010 0010	22	34		Spindle Start
0010 0011	23	35		Axis Start
0010 01xx	24–27	36–39		(null)
0010 1000	28	40		Cursor Right
0010 1001	29	41		6
0010 1010	2A	42		5
0010 1011	2B	43		(null)
0010 1100	2C	44		Page Delete
0010 1101	2D	45		S
0010 1110	2E	46		F
0010 1111	2F	47		(null)
0011 000x	30–31	48–49	ORAC Input E:	Open
0011 001x	32–33	50–51		Closed
0011 00xx	30–33	48–51	Tool Changer:	Down + Retract
0011 01xx	34–37	52–55		To Be Avoided
0011 10xx	38–3B	56–59		Up + Retract
0011 11xx	3C–3F	60–63		Up + Extend

3.4.5 Status Signals

The input signals necessary for monitoring the status of the ORAC/Z8671 interface are connected to four external 8–bit input ports. These ports are mapped to the top four bytes of the Z8671's program memory space, ie. FFFC to FFFF hex. As shown in Figure 3.13, each port is comprised of a TTL SN74ALS244 octal tri-state buffer with a 4.7K active pull-up resistor on each input pin. The output side of each buffer is hardwired OR'ed to the Z8671's multiplexed address/data bus. Individual ports are selected for reading by mutually exclusive control signals decoded from the full 16–bit address bus and the R/$\overline{W}$ line.

The lower 4 bits of input Port FFFF hex are wired to the four auxiliary output relays of the ORAC lathe. Closing a relay switches the corresponding input bit to ground, resulting in a logic 0 being read. An open relay is read as a logic 1 due to the pull-up resistors. These outputs are used by ORAC programs to inform the Z8671 when a tool change is required, what tool is required, and when the last page of a program is being executed.

Input Port FFFE hex is used to read the settings of the five micro-switches on the tool changer. The position of the rotate and clamp pistons are read on input bits 5 and 4, respectively. The three switches providing the binary code for the turret position are read on input bits 0–2. These three switches provide a number ranging from 0 to 7. The tool turret, however, has its positions marked as 1 to 8. To correct this anomaly, input bit 4 is wired as the binary NAND of bits 0–3. This allows the lower 4 bits of input Port FFFE hex to be read as the complement of the current tool turret position. A complete truth table for all possible 8–bit input values is provided in Table 3.3.

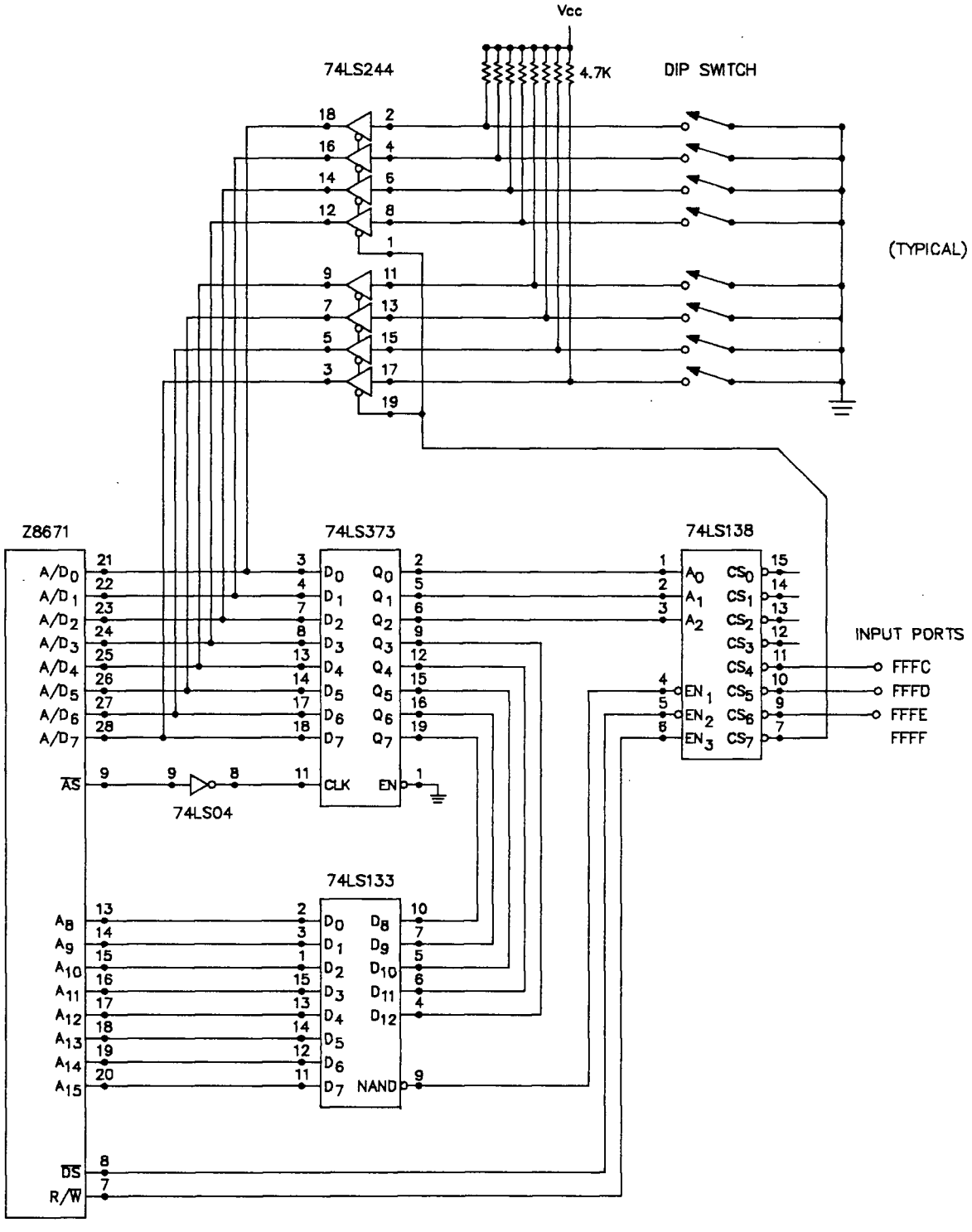

Figure 3.13: Address Decoding Circuit for Input Ports FFFC to FFFF

Table 3.3: Tool Changer Position Decoding

Input Value Read at FFFE			Pistons		Turret
Binary	Hex	Decimal	Clamp	Rotate	Position
1100 xxxx	C0-CF	192-207	Clamped (Error Condition)	Extended	x
1101 0111	D7	215	Raised	Extended	0
1110 0111	E7	231	Clamped	Retracted	8
1110 1000	E8	232	Clamped	Retracted	7
1110 1001	E9	233	Clamped	Retracted	6
1110 1010	EA	234	Clamped	Retracted	5
1110 1011	EB	235	Clamped	Retracted	4
1110 1100	EC	236	Clamped	Retracted	3
1110 1101	ED	237	Clamped	Retracted	2
1110 1110	EE	238	Clamped	Retracted	1
1111 0111	F7	247	Raised	Retracted	0

The 8 input lines of memory mapped Port FFFD hex are individually connected to system ground through an 8 pole DIP switch located on the interface board. The BASIC/DEBUG interpreter reads the position of the first three switches once upon start-up or reset to determine the baud rate to be used for serial communication. The remaining five switches are not used. The required switch settings for the eight available baud rates are listed in Table 3.4. In order to accommodate the multiplexing of ORAC NC programs and Z8671 commands, the baud rate should be set to 2400, as this is the fastest rate at which the lathe can accept data.

Input Port FFFC hex is used to verify the switching action of the 22102 cross point arrays. This is done by checking to see if the appropriate ORAC signal line gets pulled to ground when a particular cross point switch is activated. A logic 0 read on input

Table 3.4: Z8671 Baud Rate Switch Settings

Baud Rate	Switch 1	Switch 2	Switch 3
110	CLOSED	OPEN	OPEN
150	CLOSED	CLOSED	CLOSED
300	OPEN	OPEN	OPEN
1200	OPEN	CLOSED	OPEN
2400	CLOSED	CLOSED	OPEN
4800	OPEN	OPEN	CLOSED
9600	CLOSED	OPEN	CLOSED
19200	OPEN	CLOSED	CLOSED

bits 0–4 indicates a switch closure in columns 1–5, respectively, of the data entry section of the ORAC keypad. Bits 5 and 6 provide confirmation of the Axis Stop and Axis Start switches.

Verification of the Spindle Start and Spindle Stop switches is obtained directly from the spindle's optical encoder. As shown in Figure 3.14, the 48 pulse/revolution output from the encoder is used to trigger a CMOS 14538 precision monostable multivibrator. This multivibrator is configured for retriggerable operation with a time constant of 161.5 ms. This results in a constant logic level 1 output when the spindle is operating at 10 rpm or greater, and a constant logic level 0 output when the spindle is stopped.

The multivibrator signal and its complement are read by the Z8671 MCU on bits 2 and 3 of I/O Port 3. In addition to being normal inputs, these bits are configured as two maskable, negative edge-triggered, hardware interrupts. By selectively enabling these interrupts, the Z8671 software can monitor the spindle for an unexpected change of state, either starting or stopping. In each case, the interrupt handling routine will generate an

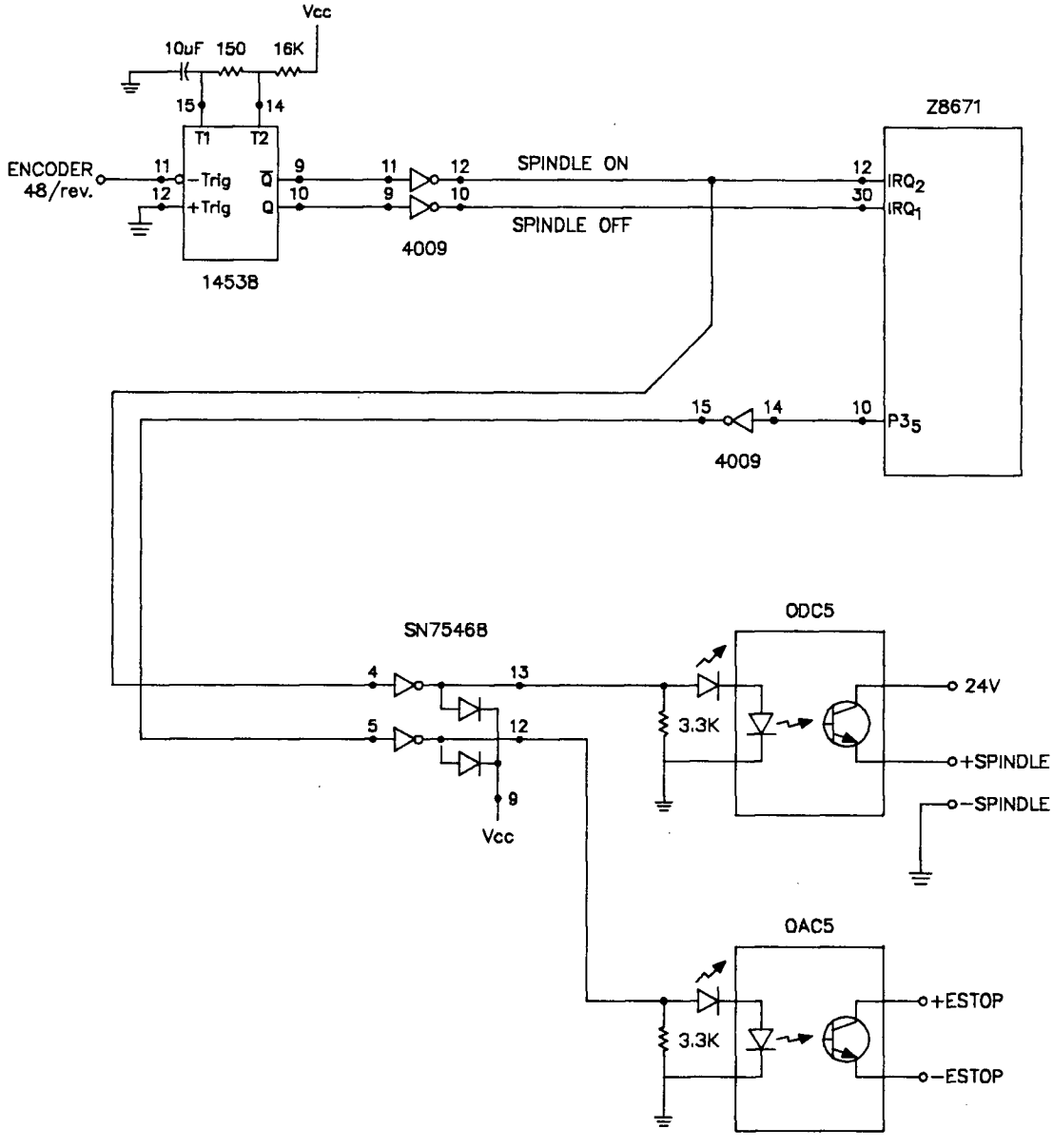

Figure 3.14: Spindle Monitoring Circuit

emergency stop on the ORAC lathe to minimize injury and damage. The status of the ORAC spindle is also made available to external devices through an optically isolated ODC5 output module. A 24 V DC signal indicates that the spindle is running; an open circuit indicates that the spindle is stopped.

3.5 Software Interface

The overall software interface between the VAX and the ORAC CNC lathe is comprised of two cooperating components; one executing on the ZILOG Z8671 microcomputer, the other executing as a subprocess on the VAX 11/750 computer. The Z8671 component provides the interface to the hardware, and is responsible for the low level control and status monitoring. The VAX component provides the interface to both the operator, and the supervisory level flexible manufacturing software. It is responsible for translating the commands received from the supervisory software into the low level functions provided by the Z8671 software, initiating the execution of these functions, and reporting the completion status back to the supervisory software.

3.5.1 Z8671 Software

3.5.1.1 Languages and Memory Requirements

The Z8671 component of the VAX to ORAC software interface is written in both BASIC/DEBUG and Z8 machine language. BASIC/DEBUG is a subset of standard Darthmouth BASIC and recognizes a total of 14 commands and 3 functions as outlined

Table 3.5: BASIC/DEBUG Keywords and Operators

Commands:	GO@	GOSUB	GOTO
	IF/THEN	INPUT	IN
	LET	LIST	NEW
	PRINT	REM	RETURN
	RUN	STOP	
Functions:	AND()	HEX()	USR()
Operators:			
Arithmetic:	+ - / *		
	\ [unsigned division]		
	% [hexadecimal constant]		
Relational:	< <= = <> >= >		
Memory:	@ [8-bit memory reference]		
	~ [16-bit memory reference]		

in Table 3.5. BASIC/DEBUG also supports 6 arithmetic operations; addition, subtraction, multiplication, signed and unsigned division, and bitwise logical AND. These operations are supported for integers only. Character and floating point data types are not supported. A total of 26 16–bit named variables (A–Z) are supported. In addition to variables and constants, any 8 or 16–bit memory location may be referenced within a numeric expression.

Routines written in Z8 machine language may be called from within BASIC/DEBUG as either subroutines or functions by using the GO@ command and USR function, respectively. The BASIC/DEBUG interpreter does not process interrupts. Interrupts are

vectored through the Z8671 hardware to external memory locations 1000–1011 hex. In order to process interrupts, these locations must contain Z8 JUMP instructions to the appropriate machine language routines.

The BASIC/DEBUG program necessary to monitor and control the ORAC lathe hardware occupies 2.95K of memory. In addition to the standard BASIC/DEBUG functions and variables, this program utilizes seven Z8 machine language routines, and a 46 byte data array implemented in external memory. Five of the Z8 machine language routines provide binary operations not supported by BASIC/DEBUG. These include 8 and 16–bit binary complement, the setting and clearing of selected bits within a 16–bit value, and the reading of selected bits from an 8–bit input port. The sixth Z8 function waits until a specified ASCII character is received on the serial input line. The remaining Z8 machine language routine is an interrupt handler which activates the ORAC emergency stop output in response to an unexpected state change of the spindle. The data array is used to store the output and verification codes for each of the 23 keys on the ORAC keypad which are accessible through the hardware.

Routines written in Z8 machine language are referenced in BASIC/DEBUG by direct memory address. To ensure proper operation, each program segment must be loaded into external memory at the correct address. The BASIC/DEBUG interpreter does not maintain any pointers for the machine language code, so it is up to the programmer to ensure that these routines are not overwritten by either the BASIC program or the GOSUB stack. Figures 3.15 and 3.16 give a complete memory map showing the relationship between each section of code and the memory space used by BASIC/DEBUG for both the 4K RAM development mode, and the 4K ROM plus 2K RAM automatic start-up mode. It should be noted that the 4K RAM development mode utilizes non-standard

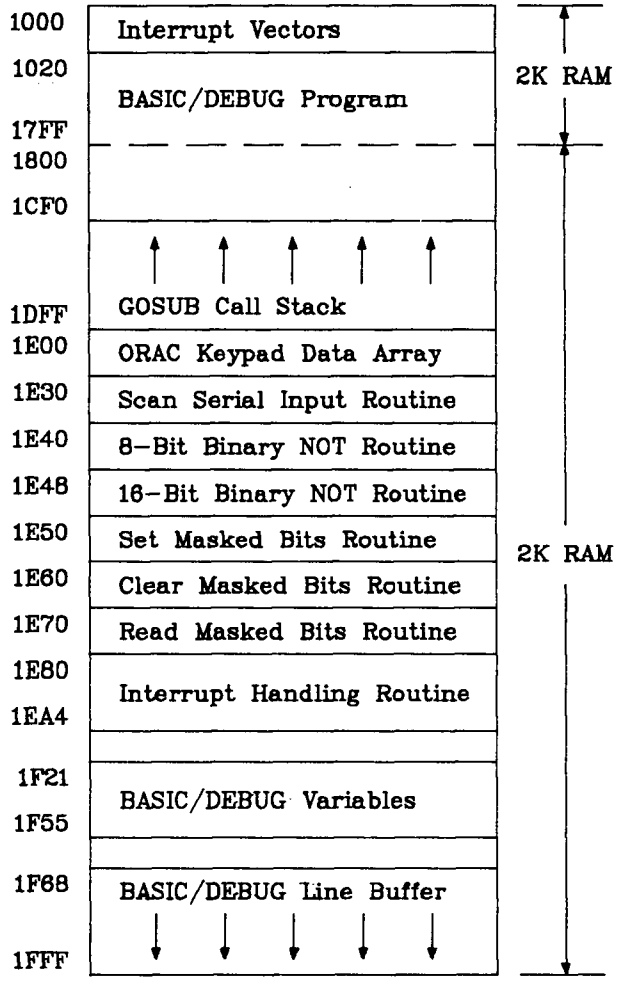

Figure 3.15: Memory Map for 4K RAM Development Mode

locations for both the beginning of the BASIC program, and the base of the GOSUB stack. The corresponding BASIC/DEBUG pointers must therefore be adjusted prior to loading the software.

The VAX software component supports both modes of operation. If the Z8671 MCU does not enter the automatic start-up mode during the initialization procedure, the VAX software will attempt to download the BASIC/DEBUG program, ORAC key codes, and

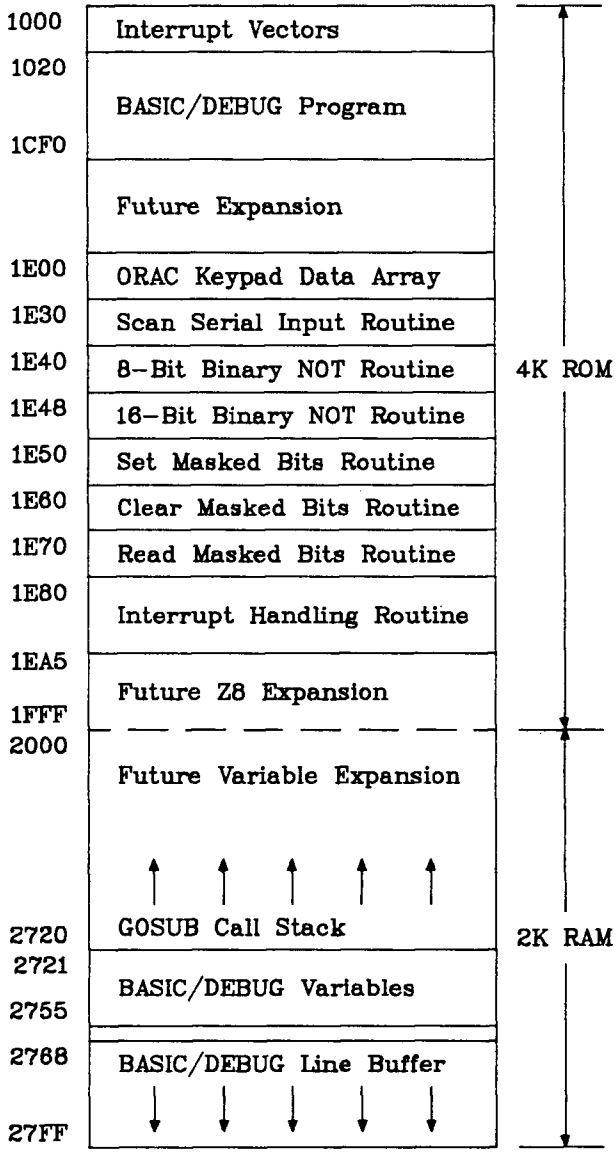

Figure 3.16: Memory Map for 4K ROM Automatic Mode

Z8 machine language routines from the VAX text file ORAC_CNTRL.BAS. This file contains the appropriate load addresses and may be used as a starting point for future development work or as a guide for generating new EPROM's for automatic start-up. (Note: This file has been written for use with the download facility provided by the VAX component of the software, and contains syntax not supported by BASIC/DEBUG. These elements have been added to improve readability and to guide the download facility. They are stripped before the program is sent to the BASIC/DEBUG monitor.)

3.5.1.2 Communication Protocol

As noted earlier, communications between the Z8671 software and the VAX computer take place over a full-duplex RS-232C serial line driven by the Z8671's onboard UART. This communication is a two way process with the Z8671 software sending both status information and prompts to the VAX, and the VAX software sending both commands and parameters to the Z8671. As with any two-way communication, it was necessary to develop a protocol to coordinate the exchange and translation of the various messages. The nature of this protocol was largely dictated by the serial I/O capabilities of the BASIC/DEBUG interpreter.

Although the Z8671 MCU is capable of generating an interrupt whenever a serial character is received or transmitted, this feature is not supported by the BASIC/DEBUG interpreter. Instead, it operates the UART in a polling mode, waiting on each input and output character as needed. The input line buffer is only used to store a single line of text—delimited by a carriage return—which is received following the execution of an input instruction. Characters which were received by the UART prior to the input instruction are ignored. This lack of a true type-ahead buffer dictates that each

message exchange must be initiated by the Z8671 software. The VAX software plays the subservient role; simply responding to prompts from the Z8671 software. The protocol also stipulates that each transmission from the Z8671 must be acknowledged by the VAX before another exchange can be initiated.

As noted earlier, BASIC/DEBUG does not support the character data type. Therefore, the standard serial input commands, IN and INPUT, only accept ASCII coded integers as valid input. In order to read non-integer input, BASIC/DEBUG programs must directly access the Z8 machine language serial input driver located at address 61 hex in internal ROM. This driver reads a single byte of input each time it is called, and returns the value as a binary number without any ASCII translation. This option was selected for the VAX to Z8671 portion of the communication protocol to facilitate more meaningful messages than could be obtained with strictly numerical input. Each transmission from the VAX is therefore only one character long.

The BASIC/DEBUG interpreter only supports one serial I/O channel. This means that the Z8671 to VAX portion of the protocol must take into account the messages transmitted by the BASIC/DEBUG monitor as well as those from the Z8671 software. All error and informational messages transmitted by the BASIC/DEBUG monitor are followed by the three character system prompt, "<CR><LF>:". Because the <CR> and <LF> precede the colon—which is the most significant character—they are not suitable delimiters if the source of the message is to be easily identified. For this reason, the Z8671 to VAX communication protocol ignores all carriage control characters and uses three different printable characters to both delimit and identify the three different message types. All transmissions ending with a colon are treated as messages from the Z8671 monitor. Messages terminating with a question mark signify a prompt for information

from the Z8671 software. A number sign is used to terminate all status messages from the Z8671 software.

3.5.1.3 Functional Overview

The BASIC/DEBUG software is comprised of an initialization header, a command monitor, the routines necessary to perform each command, and an error handler. The basic operation of the software is summarized in Figure 3.17. The initialization header first configures the Z8671 I/O architecture (as outlined in the hardware description) using the internal status and control registers, and then sets all output devices to the required initial state. The current states of the tool changer and spindle are then read and reported to the VAX software.

The command monitor informs the VAX software that it is ready to accept a command by issuing the Command? prompt, and then waits for a response. Upon reception of a valid command, a subroutine call is made to the program segment responsible for executing that command. Each program segment is responsible for initiating and monitoring a complete operation on the ORAC lathe; each starting and ending with the ORAC CNC microprocessor at the top level of its menu-driven interface. Functional support for these operations is provided by a hierarchial subroutine structure. The low level subroutines provide such universal functions as; delaying program execution for a given period of time, writing a specified value to output Port 2, turning an output module on or off, and pressing an individual key on the ORAC keypad. The middle level routines provide such combined functions as updating the system status variables and changing the tool turret to a specified position.

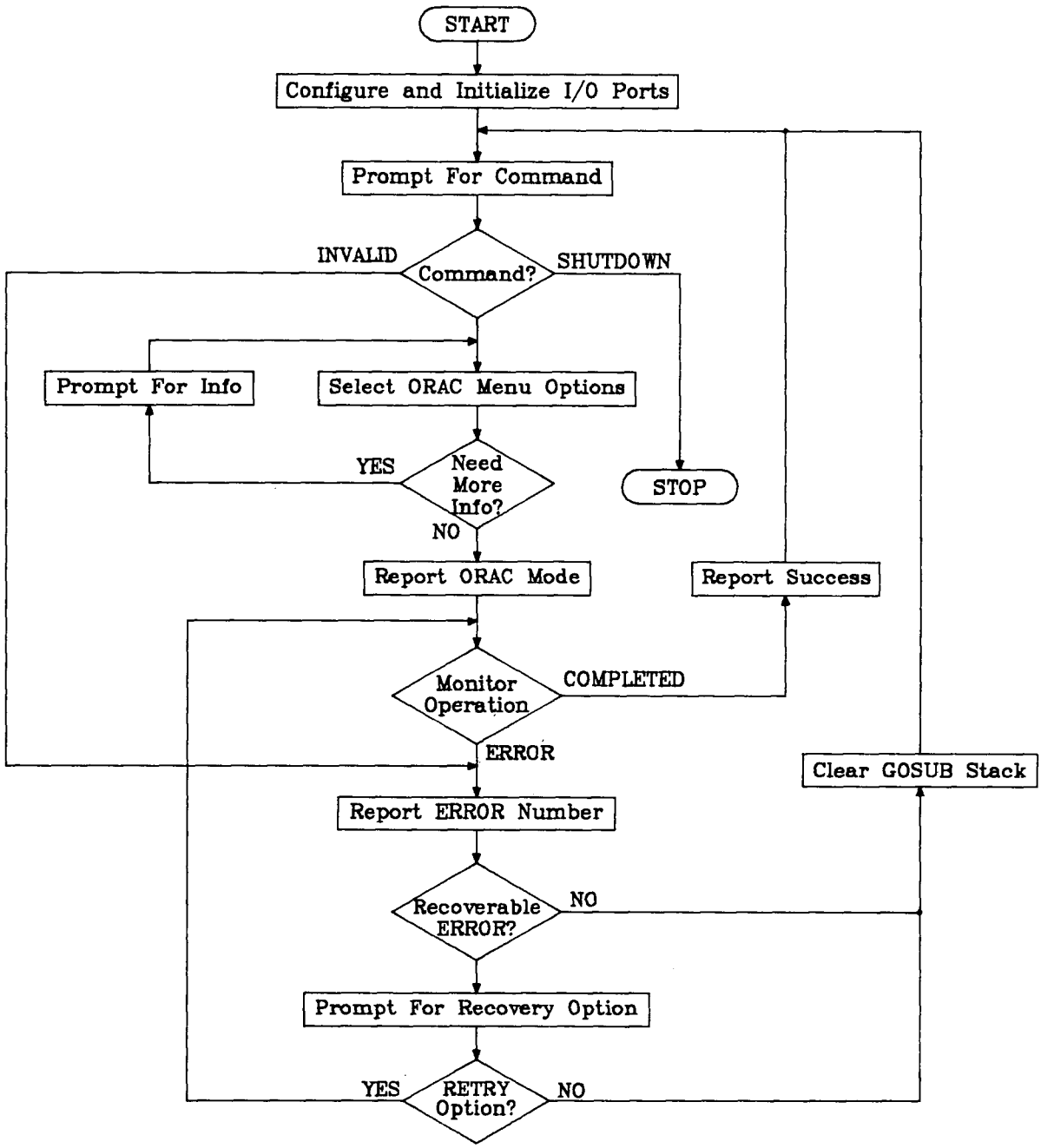

Figure 3.17: Flowchart of Z8671 Sotware Component

The first action taken by each program segment is the selection of the appropriate options from the ORAC menu structure. If a particular menu selection requires additional information, it is requested with a prompt string unique to that operation. When all menu selections have been successfully executed, the VAX software is informed of the current mode of the ORAC lathe as a check on the command translation. The Z8671 software then sets the necessary control outputs, and monitors the corresponding input signals. Monitoring continues until either the operation completes successfully, or an error condition is detected. If successful, program execution is returned to the command monitor which sends the Finished# message to the VAX software before requesting the next command. If an error is detected, program execution branches to the error handler.

There are two types of error conditions that may occur during a given ORAC operation. Non-recoverable errors, such as an emergency stop, require the ORAC lathe to be reset to the main menu. Recoverable error conditions, such as a sticky solenoid valve or low air pressure, may be cleared without resetting the lathe. Each subroutine which detects an error is responsible for setting both an error number and a recovery address before branching to the error handler. Non-recoverable errors are indicated by setting the recovery address to zero.

The Z8671 software reports each error to the VAX by numeric reference rather than by a descriptive message. The error number is then translated into the corresponding error message by the VAX software, which is not as constrained by memory limits. If the corresponding recovery address is non-zero, the error handler sends the Retry? prompt to the VAX software. If the VAX software responds with a Y, program control is branched back to the specified recovery address and normal monitoring continues. If the VAX software responds with a N, or if the error is non-recoverable, the current operation is

aborted. The error number variable is set to zero, the subroutine call stack is cleared, and execution resumes at the first line of the command monitor.

In addition to the error monitoring performed by software, the spindle status is constantly monitored by the Z8671 interrupt hardware. A maskable hardware interrupt request, either IRQ_0 or IRQ_1, is generated each time the spindle changes state. The Z8671 software disables these requests whenever a state change is expected, such as when the spindle Start or Stop key is being pressed, and enables them when the required state has been achieved. If a state change occurs while the interrupts are enabled, program control is vectored to the Z8 machine language interrupt handler loaded at address 1E80 hex. This routine sets bit $P3_5$ to generate an ORAC emergency stop condition which immediately stops both the spindle and the X and Z axes. The interrupt routine disable the spindle monitoring function just prior to returning control to the BASIC/DEBUG software. This allows manual recovery without generating a second emergency stop condition.

This hardware monitoring of the spindle status provides the shortest possible reaction time for generating the emergency stop, but it also means that the BASIC/DEBUG software is not immediately aware that anything has happened. To ensure that these errors get trapped and correctly reported to the VAX software, each of the low level functions checks the status of the $P3_5$ output bit prior to performing any I/O operations. The status of this output is also polled along with the appropriate input signals whenever the BASIC/DEBUG software is waiting for a state change.

3.5.1.4 Command Vocabulary

The Z8671 software presently recognizes a total of 4 single-character commands which perform the following functions:

M Permit Manual operation of the spindle.

L Load NC programs to the ORAC lathe.

E Execute the NC program currently loaded.

S Prepare the system for Shutdown.

Each of these commands result in a further dialogue between the Z8671 and the VAX and are described in the following paragraphs.

Manual Command The Z8671 software first disables the interrupt monitoring of the spindle, sends the Manual# confirmation message, and then waits for a response to the Continue? prompt. Whenever a character is received, the Spindle Stop key is pressed and interrupt monitoring is restarted as soon as the spindle comes to rest. The received input is not translated, therefore any character will have the same effect. Program execution is then returned to the command monitor, so the ORAC lathe must be left at the main menu.

Load Command The Z8671 software makes the appropriate ORAC menu selections to prepare the lathe for program reception at 2400 baud. The tri-state buffer allowing the ORAC lathe to listen in on the serial transmissions from the VAX computer is then

enabled. The confirmation message LOADING#, is sent to the VAX to indicate that serial transmission of the NC program may be started.

Each character of the NC program is read by the Z8671 software using the single-character input driver. The echo provided by the input driver is used by the VAX software to check for serial transmission errors. Each received character is checked for an ASCII control D which signifies the end of the NC program. This comparison is performed in Z8 machine language to ensure the fastest possible transmission rate. Program control is then passed back to the command monitor.

Execute Command The appropriate menu selections are made to bring the ORAC lathe to the Tool Offset submenu. The VAX software is then prompted for one of three possible options.

Q The Quit option causes the Z8671 software to exit the Tool Offset menu, leaving both the floating zero point and the tool offsets at their current settings.

M The Manual option disables spindle monitoring to allow the operator to manually set the floating zero point and/or tool offsets. As with the Manual command, execution continues whenever a serial input character is received. Unlike the Manual command, however, the lathe must be returned to the Tool Offsets submenu rather than the Main menu.

Z The Zero option is used to automatically reset the floating zero point. The Z8671 software will press the appropriate keys to cause the ORAC microprocessor to accept the current position as 20.0 mm in X, and 0.0 mm in Z, with zero offset.

After the selected Tool Offset option has been completed, the Z8671 software presses the ORAC F key to exit the Tool Offset menu and start execution of the NC program. During the machining sequence, the lathe's auxiliary I/O signals provide the only communication path between the ORAC microprocessor and the Z8671 MCU. The NC part program must contain the correct sequence of I/O statements to inform the Z8671 software whenever a new tool is required and when the last program statement is about to be executed.

To ensure proper synchronization between the NC program and the Z8671 software, a simple request/acknowledge handshake protocol requiring three ORAC Auxiliary I/O statements is used for each data transfer. The NC program must first request the attention of the Z8671 software by setting all four auxiliary outputs to a logic 1, and then wait for an acknowledgement on auxiliary input E. The Z8671 software detects this request during its periodic polling of input Port FFFF hex, and acknowledges reception of the request by providing a 24 V signal to auxiliary input E. The Z8671 software then waits for the NC part program to write a new value to its auxiliary output port. If the new value represents a valid tool number, ie. 1–8, it is interpreted by the Z8671 software as a request for a new tool. If the new value is 14, the Z8671 software assumes that the end of the machining process has been reached.

In addition to the correct protocol, the NC program must follow certain guidelines as to the placement of these auxiliary I/O statements. Each program which includes an ORAC machining instruction must contain at least one tool change request. This request must be placed directly between the Program Datum instruction and the first machining instruction. This placement ensures that the tool changer is at a safe location when the Z8671 software activates the pneumatic pistons. For each subsequent tool change, the NC

program is responsible for positioning the tool holder at a safe location, preferably the program datum, prior to the tool change request. In order to ensure proper machining of subsequent parts, the NC program must specify a tool change to tool position 1 and move back to the program datum prior to sending the End of Program code.

Whenever a tool change request is detected, the Z8671 software first ensures that the spindle and axes are stopped by pressing the appropriate keys on the ORAC keypad. If the turret is not already in the required position, the pneumatic pistons are activated to reposition the turret. The spindle and axis start keys are then pressed to resume program execution, and the current tool position is reported to the VAX software. When the End of Program code is detected, program execution is returned to the command monitor.

Shutdown Command This command is issued by the VAX software to signify that it no longer requires access to the ORAC interface. The Z8671 prepares the lathe for shutdown by turning off the spindle, stopping the X and Z axes, and clamping the tool changer in its current position. All output modules and cross point switches are turned off just prior to termination of the control program.

3.5.1.5 Communication Summary

The syntax and expected response for all messages used by the Z8671 software is provided in Table 3.6. The syntax of a generic BASIC/DEBUG error message is also provided. Items enclosed in braces indicate that one item from the specified range or list of items is to be substituted. A single character preceded by a caret is used to represent

Table 3.6: Z8671/VAX Communication Summary

From Z8671 MCU			From VAX 11/750
Tool={1–8}#			<CR>
Spindle={0|1}#			<CR>
Command?			{M|L|E|S}
[M]	Manual#		<CR>
	Continue?		<CR>
[L]	Loading#		<CR>
[E]	Offsets?		{M|Q|Z}
	[M]	Manual#	<CR>
		Continue?	<CR>
	Machining#		<CR>
	Tool={1–8}#		<CR>
	Spindle={0|1}#		<CR>
[S]	<CR><LF>:		
Finished#			<CR>
or			
Error={1–18}#			<CR>
Retry?			{Y|N}

<error>!^G AT <line><CR><LF>:

the specified control character, however, the standard carriage control characters, ^J and ^M, are written in their more common forms, <CR> and <LF>.

The relative timing of each message is indicated in Table 3.6 by its position. Messages which are single spaced always follow one another in the specified order. Messages which occur in response to a previous prompt are indented to show their dependence. When more than one response is possible, the dependant messages are grouped together and preceded by the corresponding response enclosed in square brackets.

3.5.2 VAX Software

The VAX component of the VAX/ORAC software interface is contained in an executable image named ORAC_driver.exe. This module serves as a middleman between the low level Z8671 software and the upper levels of the software architecture. It is responsible for establishing communications with the Z8671 MCU and for keeping track of the current state of both the Z8671 and the ORAC lathe. It is also responsible for establishing communications with both the operator and the supervisory software using the standard protocols developed for this purpose.

Functionally, this component of the software is similar to the Z8671 software; incorporating an initialization sequence, a command monitor, a module for monitoring and reporting status information, and an error handler. The data channels used by each segment, however, are different from those used by the corresponding Z8671 segments. A pictorial description of the various data channels used by the VAX component of the ORAC interface software is presented in Figure 3.18. In this diagram the parent process

is the FMC supervisory software and the ORAC driver software has been executed with the subprocess name ORAC1.

3.5.2.1 Communications with the Z8671 MCU

The VAX software communicates with the Z8671 over one of the standard terminal lines provided by the VAX 11/750 hardware. The only stipulation placed on the choice of a terminal line is that it be a direct connection configured for non-network communications. (Serial lines that are connected through a terminal server may not be used because the corresponding virtual device names are *floating* and get incremented after each session, making it difficult for these lines to be allocated by application programs.) The other necessary characteristics, such as baud rate and type-ahead buffer size, are set by the VAX software after the terminal line has been successfully allocated.

The protocol developed for communications with the Z8671 requires the VAX software to handle asynchronous messages of varying length with nonstandard terminators. The standard I/O interface provided by FORTRAN isn't capable of providing this functionality, so the ORAC driver module makes use of the VAX system service routines to directly access the full I/O capabilities of the VMS terminal driver. These capabilities include a 2K type-ahead buffer, user defined message terminators, and interrupt handling of asynchronous events. The terminal driver is analogous to a telephone answering service; dividing the incoming characters into complete messages terminated by either a colon, a question mark, or a number sign, and storing them until read by the ORAC driver software.

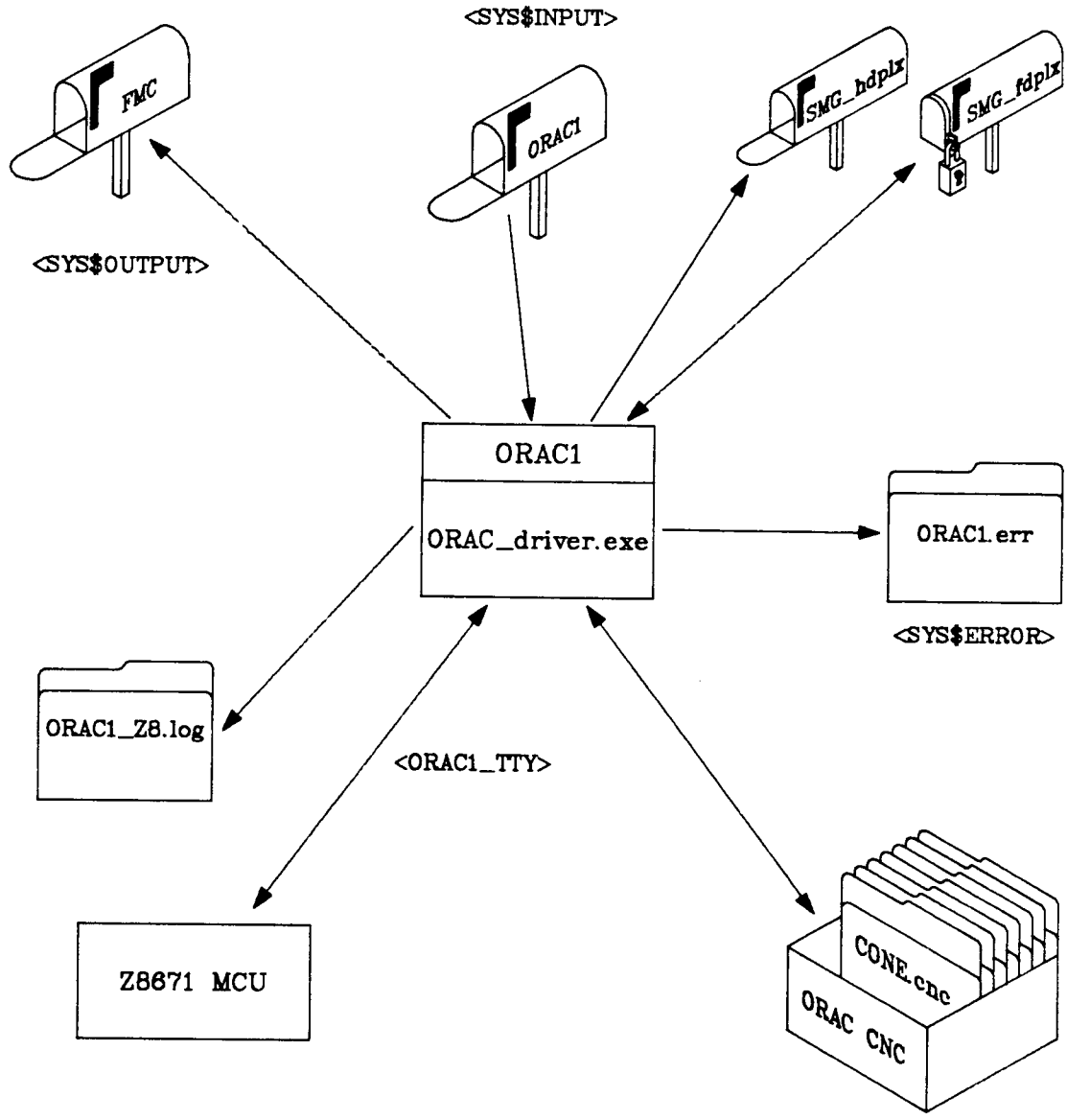

Figure 3.18: Data Channels used by ORAC Driver Module

Since the Z8671 input driver echoes all received characters, the actual messages from the Z8671 software will be interspersed between characters transmitted by the VAX. To ensure that these characters do not interfere with the proper translation of the Z8671 messages, they are pulled from the buffer as soon as they received. This allows the routine which writes to the Z8671 to do an error check on the transmission, and also ensures that characters are not overwritten before the Z8671 software has a chance to read them.

The ORAC driver software accesses the type-ahead buffer in both the interrupt and polled modes. The interrupt mode is used whenever the driver is busy performing other tasks, or when the time between messages is not precisely known. The *wait with timeout* mode is used whenever a response is expected within a known timeframe, such as when waiting for the Command? prompt after receiving notification that the previously issued command has successfully completed. This technique maximizes the software's ability to detect a failure on the serial line or a loss of synchronization.

The syntax of each message read from the type-ahead buffer is checked using the same subroutine, regardless of the access mode used to read it. The termination character is first checked to determine the message type. The text of the message is then parsed from left to right and compared to the keywords recognized for that message type. If the syntax is valid, the appropriate state variables for the Z8671 MCU, the ORAC lathe, the tool changer, and/or the spindle are updated and displayed on the screen. An invalid message is indicated by setting the Z8671 state variable to the Z8_MSG_ERR condition. These state variables are monitored by the calling routines to ensure that the communication protocol outlined in Table 3.6 is not violated.

All transmissions which take place over the serial line are recorded in a permanent disk file to help trace any communication problems which may arise. The one exception being

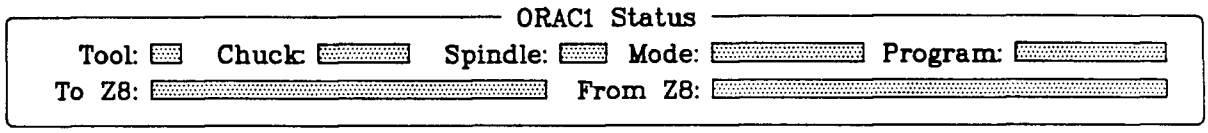

Figure 3.19: ORAC Driver Status Window

that characters echoed back by the Z8671 are not recorded unless an error is detected. Each message is preceded by an arrow indicating the direction of travel; a right-hand arrow, ==>, denotes a message from the Z8671 software, and a left-hand arrow, <==, denotes a message from the VAX software. The filename is comprised of the subprocess name appended with the characters "_Z8.log" to allow easy identification of the source in a multi-machine environment.

3.5.2.2 Initialization Sequence

In order to communicate with the various components of the supervisory system the ORAC driver module must first assign a channel number to each of the I/O devices using the process specific logical names created by the parent process. When all data channels have been successfully assigned the `Startup` message is sent to the SYS$OUTPUT device to notify the parent process that the driver is executing properly. The ORAC driver then sends a message to the SMG_hdplx mailbox requesting that the SMG process create a status window on its behalf. This window contains the 7 subwindows shown in Figure 3.19. The actual name of the ORAC driver subprocess is substituted in the label field when it is created, and the SMG process is requested to place the window in the next available location.

Each subwindow (with the exception of the Chuck subwindow) is filled in during the ensuing power-up sequence. The operator is first instructed to power on or reset the Z8671 controller. This should be performed with both the ORAC lathe and the external 24 V power supply turned off because the initial state of both the output modules and the cross point array switches is indeterminate. The ORAC driver then waits for a message from the Z8671. If no message is received within the timeout period, an Error message is sent to the parent process indicating that the ORAC interface could not be initialized. If the Z8671 enters the monitor mode, indicated by the system prompt, the ORAC driver prompts it for the upper and lower limits of RAM in the system. If the required 4K of RAM is available, the ORAC driver will download the Z8671 software through the BASIC/DEBUG monitor. A RUN command is then issued to start program execution.

When the Z8671 software is initialized and ready to accept a command, the operator is instructed to turn on the ORAC lathe and the 24 V power supply. The X and Z axes must then be calibrated by manually loading a piece of barstock and setting the initial zero point for each axis. The size of barstock used is unimportant as it will not be used to machine a part. However, the menu structure of the ORAC lathe does not allow access to the tool offset menu unless a valid program is loaded and executed. Consequently, a dummy NC program containing no machining instructions is downloaded to the lathe. The Z8671 software is then instructed to execute this program using the Manual option for setting the tool offsets.

When the ORAC lathe has been successfully advanced to the Tool Offset menu, the operator is instructed to set the floating zero point and tool offsets as necessary. As noted earlier, the floating zero is defined by setting the *offsets* for tool number zero. With the tool turret correctly loaded for the ORACAP package, the right-hand finishing

tool located in position one serves as both the reference tool and tool one. If the turret is not in the required position, the manual toggle switches may be used to reposition it.

The ORAC X axis is coincident with the spindle axis and does not change position from piece to piece. It is therefore an absolute axis and is precisely defined by the standard ORAC procedure. The standard Z axis, however, is a relative axis and its position depends on the length of the individual workpiece. To accommodate the automatic machining of randomly sized parts, an absolute reference must be obtained so that the relative axis may be correctly adjusted. This pseudo axis is calibrated by setting the initial Z axis zero point and providing the ORAC driver software with an accurate measurement of the distance between the machined face and the outermost face of the chuck jaws. (This location is the same for both sets of chuck jaws supplied with the lathe.)

Once the zero point has been set, the offsets for tools 1 through 9 must be entered into the ORAC's memory. The ORAC architecture supports the reading of these offsets from either the keyboard or the mini-cassette, but not the serial interface. Unfortunately, the mini-cassette recorder uses a parallel interface which is handled directly by the 6809 CNC microprocessor and could not be duplicated in hardware without more information on the ORAC software. The Z8671 does have access to the keyboard and could enter the tool offsets through the ORAC editing facility if a record was kept on the VAX computer. This process was found to be much slower than reading from the mini-cassette, and still required a manual transfer from the ORAC to the VAX every time a tool was replaced or repositioned. Considering the fact that the zero point has to be set manually, it was decided that it would be more efficient to have the tool offsets stored on a mini-cassette and updated as necessary.

When both the zero point and offsets have been set, the operator must leave the ORAC lathe at the Tool Offset menu so that the remaining program steps may be executed by the Z8671. The X and Z axes will move to the program datum, which is now absolutely defined, prior to termination of the program. Upon receiving confirmation from the Z8671 software, the ORAC driver sends the Ready message to the supervisory software signalling that it is ready to accept commands.

3.5.2.3 Command Vocabulary

The ORAC driver vocabulary is made up of five commands with qualifiers and parameters as outlined in Table 3.7. In keeping with the standard DEC syntax, all qualifiers are preceded by a backslash and are placed between the command keyword and the parameter list. Qualifiers which are mutually exclusive are indicated by a vertical list, with the default value being enclosed in square brackets. Parameters are indicated by a description of the required data type enclosed in angle brackets. Commands and qualifiers may be specified in either upper or lower case, and may be abbreviated to as many characters as necessary to make them unique. Individual arguments may be delimited by one or more space or tab characters.

The MANUAL and EXECUTE commands provide direct access to the equivalent single-character commands recognized by the Z8671 software. As indicated in Table 3.7, the MANUAL command does not require any parameters or qualifiers. The EXECUTE command accepts one qualifier which determines how the Offsets? prompt will be answered when it is issued by the Z8671 software. The default action is to answer with a Q; thus leaving the floating zero at its current position.

Table 3.7: ORAC Driver Command Syntax

```
MANUAL

SET_ZERO  <value>

SET_DIRECTORY  <directory>

LOAD  [/NOOVERWRITE]  <file>
      /OVERWRITE

EXECUTE  [/QUIT]
         /MANUAL
         /ZERO
```

The LOAD and SET_DIRECTORY commands provide access to the ORAC lathe's download facility. When the ORAC driver software is executed, it inherits the default directory that the parent process was initiated from. The SET_DIRECTORY command allows this default to be adjusted so that it points to the subdirectory containing the user's NC programs. Other directories may still be accessed by specifying the complete disk path in the filename parameter of the LOAD command. The ORAC driver software keeps a running record of the file currently loaded into the ORAC lathe's memory. If the same file is specified in a subsequent LOAD command the default action is to simply return to the command monitor. In the event that the program stored in the lathe is lost or corrupted, the OVERWRITE qualifier may be used to force the file to be reloaded.

The SET_ZERO command is used to reset the ORAC's floating Z axis from its current position to the one specified in the command. The required parameter is the distance

from the chuck jaws in millimeters. This command must be issued for each new part, prior to downloading the NC program for the part. The ORAC driver software first compares the position specified in the SET_ZERO command to the currently recorded Z axis position. If the difference is less than 0.01 mm—the single axis resolution of the lathe—no action is taken. If greater than or equal to 0.01 mm, the Z axis is adjusted by downloading and executing two NC programs. The first program is executed with the /QUIT qualifier and contains a single point-to-point instruction which moves the reference tool to a location whose absolute coordinates are 20.0 mm in X, and the specified value in Z. The second program is executed with the /ZERO qualifier which instructs the Z8671 software to reset the floating Z axis to the current tool location. This program doesn't contain any machining instructions, but specifies a program datum of X=35.0 and Z=50.0 to move the tool changer to a safe location.

3.5.2.4 Error Handling

The ORAC driver module is responsible for monitoring all of its data channels to ensure that the corresponding syntactical rules and communication protocols are strictly observed. It is also responsible for relaying error conditions that are detected by the Z8671 software. These conditions are generally non-fatal and may be recovered from with the operators help. Other errors, such as those detected by the VMS operating system, are non-recoverable and result in the termination of the ORAC driver process.

When reporting a recoverable error, the ORAC driver first queues up an SMG request to gain exclusive access to the shared message window. When the request is granted, the error message is printed along with the command that was executing when the error occurred. The operator is then requested to select from a number of available options.

The EXECUTE, SKIP, and ABORT options are always available, whereas the RETRY option is provided only if the Z8671 software has reporting an error that does not require the ORAC to be reset.

The EXECUTE and SKIP options cause the last command sent to the Z8671 to be either repeated or skipped, respectively. As a general rule all LOAD commands should be repeated to ensure that the NC program is free of errors when it is executed. In the case of an EXECUTE command the operator may decide to abandon the current workpiece and to proceed to the next part. Both of these options require the lathe to be reset to the Main Menu before continuing. If the operator selects the ABORT option, the ORAC driver will report the error condition to the parent process as an unrecoverable error.

Chapter 4

PUMA ROBOT INTERFACE

4.1 PUMA 560 Industrial Robot

The PUMA 560 industrial robot is a six degree of freedom, anthropomorphic, servo-controlled manipulator, with a rated payload of 2.5 kg. All six joints are revolute in nature, and have the range of motions shown in Figure 4.1. Each axis is fitted with an optical encoder and is digitally servoed by its own dedicated 6809 microprocessor, resulting in an overall rated repeatability of ± 0.1 mm.

Each axis is also equipped with a potentiometer which provides a *coarse* measure of the absolute position of the joint. These potentiometer signals have a resolution corresponding to approximately 1/4 revolution of the motor shaft. This resolution is not fine enough for precise motion control, but is sufficient to provide a reference position upon power-up. Each axis may then be calibrated by moving through one revolution until a high precision index signal is read from the incremental encoder. This combination of potentiometer and encoder signals eliminates the need to manually move the robot into a predefined calibration position upon power-up.

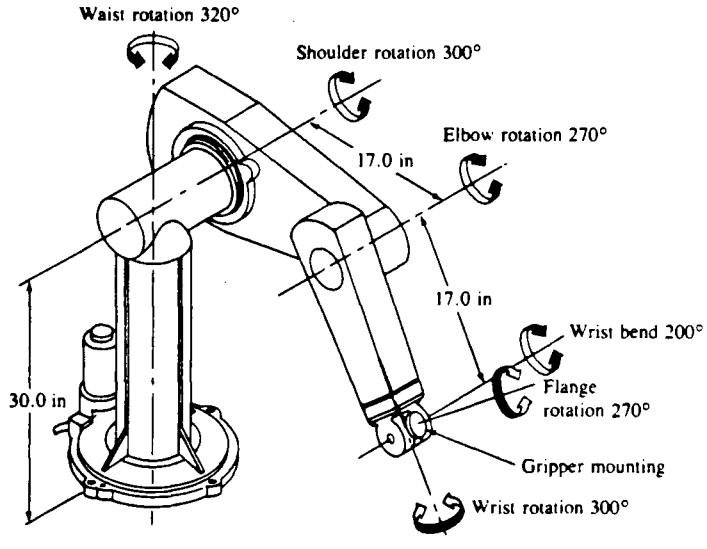

Figure 4.1: PUMA 560 Joint Configuration

The overall operation of the robot system is controlled by the VAL II operating software which executes on a DEC LSI/11 microprocessor. Robot motions may be controlled directly from the system terminal, a hand held teach pendant, or through programmed instructions. The VAL II programming language is an interpretive BASIC-like programming language which allows the robot to perform numerous tasks. This language provides a distinction between location data, numeric data, and program instructions which may all be generated independantly. All generated data is automatically stored in battery backed memory but may also be stored on floppy disk.

The VAL II operating system provides for communication over a number of different I/O channels as shown in Figure 4.2. The most advanced of these are the ALTER and SUPERVISOR channels which allow the VAL II controller to communicate with external

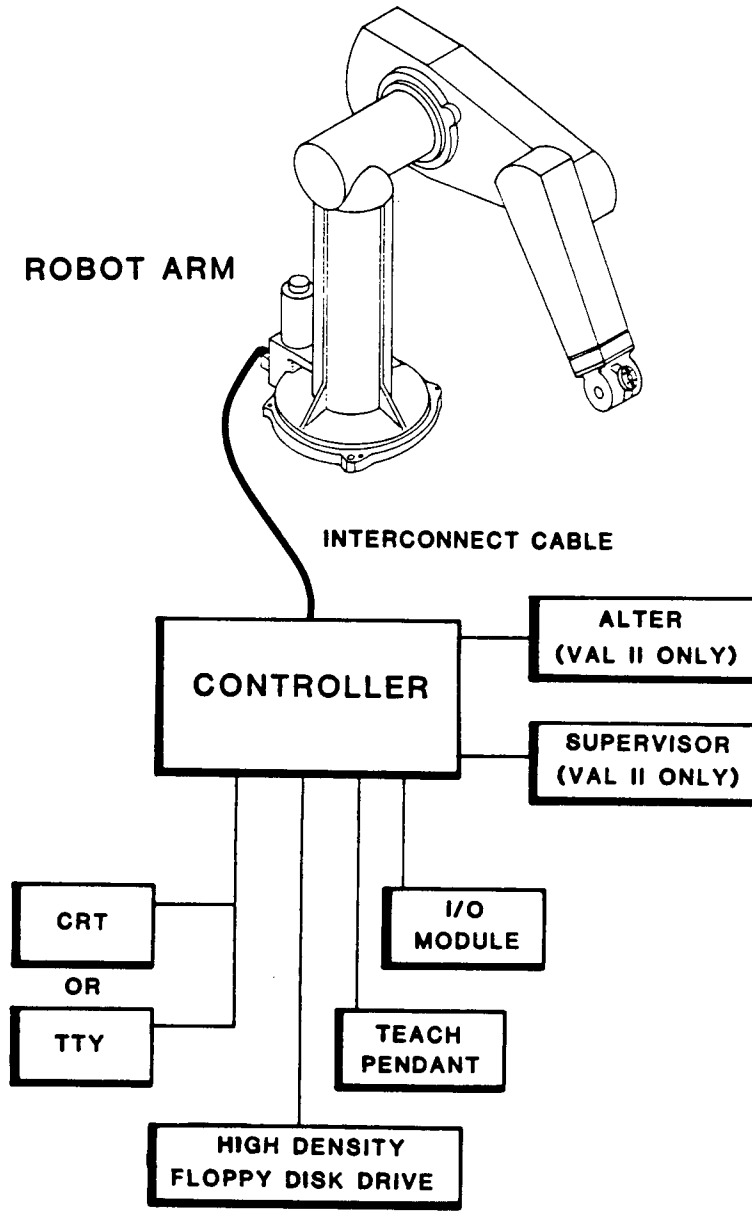

Figure 4.2: VAL II System Information Flow

computers. The Alter channel allows a previously programmed robot path to be modified in real-time based on information received from the external computer. The Supervisor channel allows an external computer to enter into a dialogue with the VAL II command monitor and directly control the overall operation of the system. It is this channel that has been utilized in the development of the interface to the VAX 11/750.

4.2 VAL II Supervisory Communication Protocol

The VAL II Supervisor interface provides a structured alternative to the standard terminal emulation mode of supervisory computer control. All communications between VAL II and the external computer are subjected to a rigorous, multi-layered communications protocol which ensures error free transmission/reception, and provides a limited set of possible responses. Message translation is further simplified by the omission of all headers and other optional text, and the substitution of numeric codes for standard informational, warning, and error messages. The use of a separate RS–423 serial line for the supervisory communications also permits both the system terminal and the disk drive unit to remain accessible during supervisory control.

The supervisory communications protocol is implemented within VAL II as a software hierarchy consisting of three distinctive layers. Each layer is responsible for implementing a specific portion of the overall protocol and correspond respectively to the Application, Transport, and Communications Subnet levels of the general communication standard adopted by the International Standards Organization (ISO) [16].

The top layer of the hierarchy is comprised of four subtasks which are responsible for managing four distinct VAL II functions; status monitoring, command translation,

Table 4.1: Logical Units used by VAL II Supervisory Interface

LUN	Data Type Transmitted	VAL II Logical Switches	
		Initialization	Data Format
0	Network manager commands	NETWORK	
1	Robot status information	(none)	
2	Synchronous monitor commands/messages	SUPERVISOR	INTERACTIVE
3	Asynchronous monitor messages	SUPERVISOR	INTERACTIVE
4	Input and output from user programs	REMOTE.PIN	MESSAGES
5	Disk commands and data transfers	DISK.NET	

program execution, and file management. Each subtask deals with a specific type of data which it receives and transmits over one of five internal data channels. These channels are assigned logical unit numbers (LUN) ranging from 1–5, and are dedicated to the specific data types indicated in Table 4.1. All data transfers are initiated by the VAL II subtasks, with the external device providing the required response.

All communications from the top layer are directed to the appropriate hardware through the middle layer protocol. Because separate data channels are used for each message type, the communications from each subtask may be directed independent of the others. During *normal* operations, the first three subtasks have their communications directed to the system terminal, while communications from the fourth subtask are directed to the local disk drive unit. When supervisory control is enabled—by setting the VAL II NETWORK logical switch to on—any combination of logical units 2–5 may be dynamically selected for communications over the network channel by enabling the corresponding switches indicated in Table 4.1.

When selected for network operations, communications between each of the four top layer tasks and the external computer are directed through the network manager. This module multiplexes the five logical data channels so that they may all talk over the single physical line. Outgoing messages are accepted via interrupt from all five logical units and are queued for serial transmission in a first-in first-out buffer. Incoming messages are accepted from the bottom layer software and routed to the proper top layer task based on the logical unit address embedded in the message.

The network manager enforces a handshaking protocol that ensures that each transmission from a VAL II logical unit is properly acknowledged by the external computer before another output request is accepted from that logical unit. This rule does not prevent other logical units from sending or receiving messages while one logical unit is waiting for its response, provided that all communications are initiated by VAL II. The supervisory computer may, however, cancel all outstanding communications by sending an abort request directly to the network manager, which is accessed as LUN 0. This request, which may be sent at any time, is not acknowledged directly by the network manager, but rather it results in an abort command being sent from each VAL II task that is waiting for input. The supervisory computer must then acknowledge each abort command as if the cancellation had been initiated by the VAL II controller. Normal communications are resumed when the VAL II monitor task requests a new supervisory command over LUN 2.

The bottom layer of the communication protocol provides an interface between the middle layer and the actual communication hardware. Its main task is to ensure that each message is transmitted and received error free. It achieves this by implementing a single-node version of the Digital Data Communications Message Protocol (DDCMP) developed

by DEC. This protocol accepts variable length input messages, and does not require any specific termination characters. Both an 8–byte header and a 16–bit polynomial Cyclic Redundancy Checksum (CRC) are appended to all messages prior to serial transmission. The CRC is capable of detecting 98% of all possible single bit transmission errors, and DDCMP automatically calls for retransmission whenever an error is detected.

The complete format of all message records transmitted over the supervisory interface is shown in Figure 4.3. The ID byte provides the address of the logical unit which is to receive the message. The 1–byte function code is used by VAL II to indicate the type of action being requested. The six valid codes and their respective actions are detailed in Table 4.2. The response from the supervisory computer should contain the same function code with the acknowledge bit (bit 7) set. The 2–byte function qualifier is logical unit dependant and is employed by VAL II to provide further control information, such as the numeric codes that replace standard text typed at the terminal. The function qualifier in records received from the supervisory computer is interpreted by VAL II as a status code indicating the success (positive) or failure (negative) of the requested action. The ASCII coded message data field is used by both VAL II and the supervisory computer to convey function dependant data which can not be numerically coded, such as command strings and file data.

4.3 VAX Software Interface

The VAX software necessary to incorporate the PUMA 560 robot into the supervisory control architecture is contained in an executable image named VAL_II_driver.exe. As

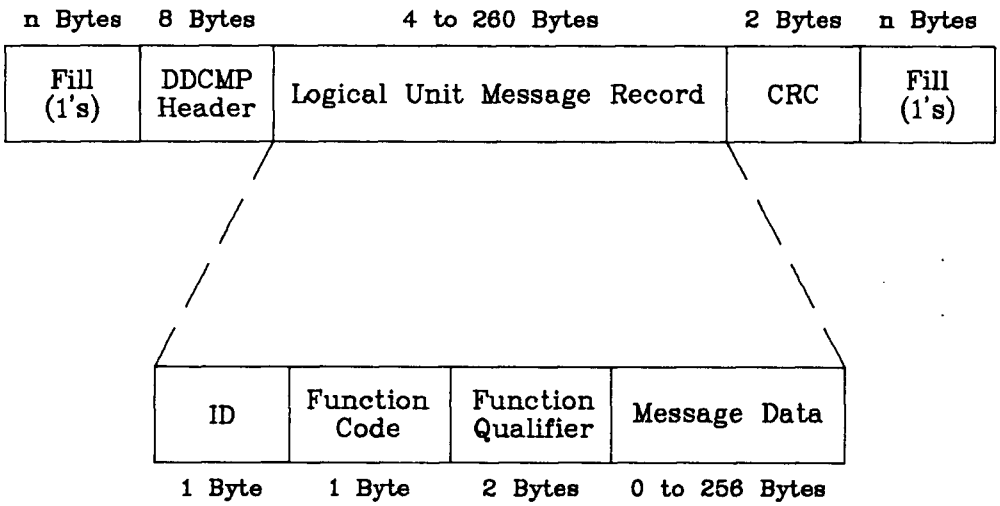

Figure 4.3: VAL II Supervisor Message Format

Table 4.2: VAL II Function Codes

Function Code	Action Requested by VAL II
0	Open/close Supervisor disk file
1	Abort outstanding communication
2	Read data from Supervisor
3	Write data to Supervisor
4	Prompt Supervisor for a reply
5	Read Supervisor disk directory

the name indicates, this module is really an interface to the VAL II supervisory communication protocol, and may therefore be used to supervise any industrial robot which utilizes the VAL II operating system. This software has the same responsibilities as the ORAC_driver.exe module, and therefore has a similar overall structure. The advanced nature of the VAL II communication protocol, however, requires a more substantial message handling capability, and allows an expanded command vocabulary.

4.3.1 Communications with the VAL II Controller

The software required to implement the DDCMP portion of the supervisory communication protocol is available on the VAX 11/750 as part of the DECnet software package. The protocol may be initiated on any valid terminal line, provided that it has previously been declared as a network device by the system manager. Once initiated, the DDCMP module is transparent to the application software, and all standard I/O routines may be used to access the I/O channel.

In compliance with the requirement for flexibility, the VAL II driver software has been written such that the terminal line used to talk to the VAL II controller may be specified at run time. Currently, the only VAX terminal line which has been set up as a network device is "_TTB5:". If the VAL II driver is to be used with other terminal lines, it is important to realize that the I/O characteristics of a VAX network device can not be altered by the application software. Consequently, the baud rate must be set to 9600 prior to declaring the terminal line a network device.

When deciding how to implement the rest of the VAL II protocol a number of factors had to be considered. Although the VAX/VMS operating system is capable of supporting

real-time computations, it was necessary for the VAX 11/750 to remain accessible as a multi-user system during both development and implementation. The large variations in the effective processing power available to each user required a trade off between the desire to keep the communication overhead low, and the need to provide sufficient buffering to ensure that information was not overwritten.

Another factor to be considered was the relative priority of the four different types of data received from the VAL II controller. The VAL II network manager accepts and transmits data from all five logical units at the same priority. This scheduling format is adequate for the VAL II side of the network because it plays the dominant role of supplying the current state, whereas the remote side merely supplies the acknowledgement.

From the supervisory computer viewpoint, the status information received from LUN 1 has the highest priority as it directly affects the sequencing of other events—including the processing of data received from other logical units. The VAL II controller transmits this data on a continual basis (as opposed to an event-driven basis) of one record every second, requiring the translation and comparison of a large number of records. The time critical nature of each record requires that the LUN 1 data be processed as quickly as possible to ensure that all decisions are based on current information.

As noted earlier, the VAX AST interrupt service only supports one level of user interrupts in a single process. The VAL II driver was therefore designed as two cooperating tasks. One task provides the standard interface to both the SMG and flexible manufacturing software, as well as the translation of all VAL II messages from logical units 2–4. The other task handles all of the direct I/O interactions with the RS–232C serial line, and serves as a combined network manager and status monitoring task. This structure

allows the LUN 1 data to be processed promptly, and also provides the buffering needed for messages received from the other logical units.

The network manager/status monitoring task is created as a subprocess by the VAL II driver software, and is therefore transparent to those images executing at higher levels of the hierarchy. As with all subprocesses, it is created with a unique name and an associated termination mailbox. The subprocess name is derived by concatenating the characters "_LUN0" to the name of the parent VAL II driver process. This convention guarantees a unique name, even if more that one VAL II controller is being supervised. The various data channels used to transfer data between these two tasks are illustrated in Figure 4.4, and are described in detail in the following paragraghs. In this illustration the VAL II driver image has been executed in the context of a subprocess named PUMA1. The network manager/status monitoring subtask is consequently named PUMA1_LUN0. In addition to the data channels shown in Figure 4.4, the VAL II driver processes also communicate with the SMG and cell management software using the standard mailbox architecture outlined in Chapter 2.

The combined network manager function reads incoming *messages* from the VAL II controller directly from the RS–232C serial line, while outgoing *responses* from the parent VAL II driver process are read from the "LUN0" mailbox. All messages and responses are read via interrupt, and are first checked to see if they comply with the handshaking protocol outlined for communication with VAL II logical units. Messages or responses which are received out of order are discarded; with the exception of abort requests, which may be received at any time. Valid *responses* are queued for serial transmission to the VAL II controller over the RS–232C line, whereas valid *messages* are routed to the appropriate function or process depending on the value specified in the logical unit field.

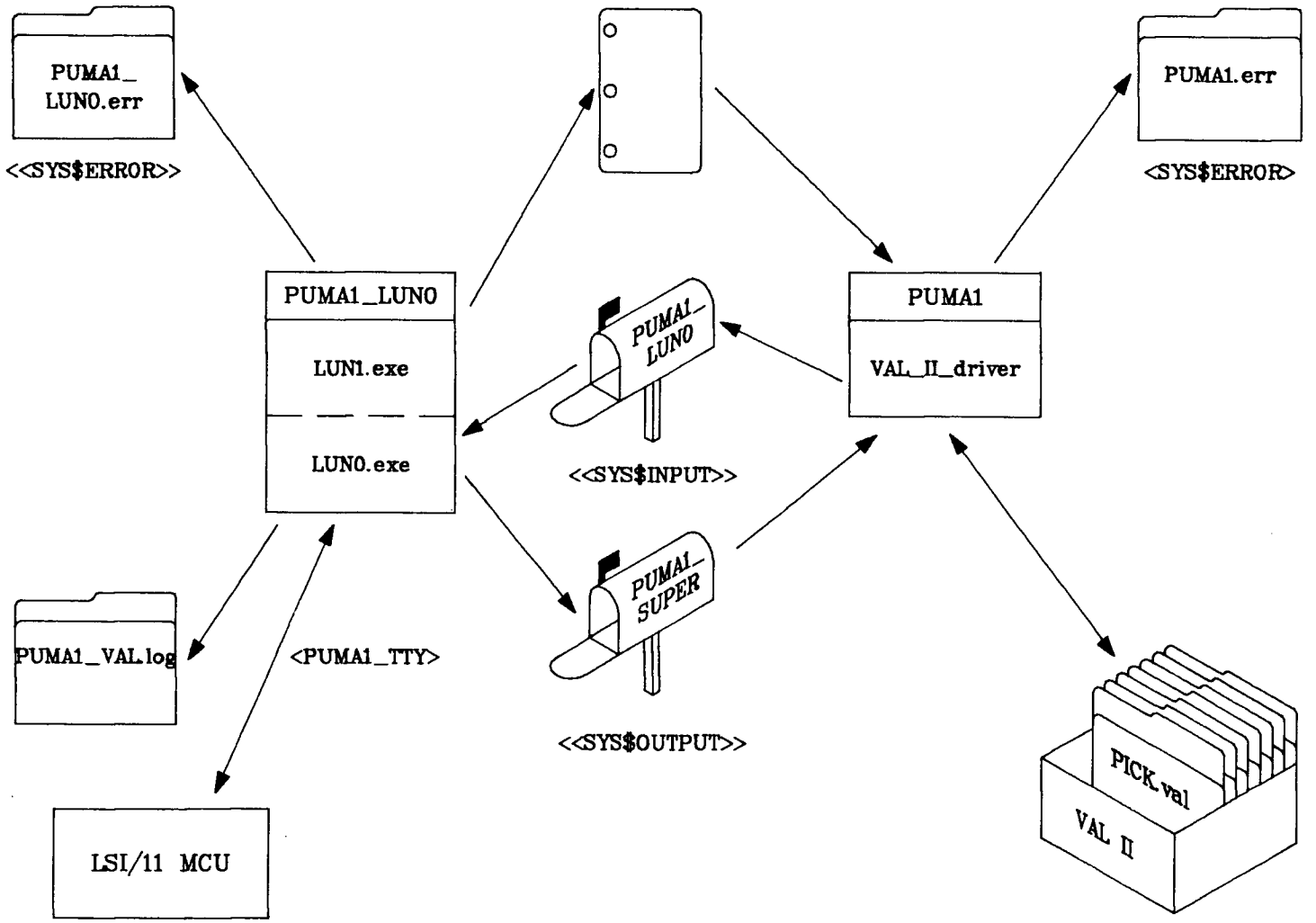

Figure 4.4: Data Channels used by VAL II Driver Subtasks

Valid LUN 1 messages are handled directly at the interrupt level by the status monitoring function of the LUN0 subprocess. The five status fields of the LUN 1 message are translated and the resulting values compared to the eight corresponding state variables. If a state change is detected, the affected variables are updated and a screen update request is sent to the SMG process via the SMG_hdplx mailbox. This update request, if required, is mailed using the *nowait* option so that the LUN0 subprocess is not unduly delayed. The required LUN 1 acknowledge record is queued for serial transmission by mailing it to the LUN0 mailbox. Program control is then returned to the main routine for servicing of the next interrupt.

As illustrated in Figure 4.4, the eight state variables updated by the LUN 0 subprocess are stored in a reserved section of memory which has been declared as a *global page section* using the VAX/VMS memory management routines. This declaration allows two or more cooperating processes to share physical memory by requesting that specified portions of their local virtual variable space be mapped to the reserved section. This technique does not support interrupt handling, such as provided by the I/O routines, but it does provide the fastest means of interprocess communication. This ensures that the VAL II driver process is always making its decisions based on the most recently available state information.

Valid VAL II messages received from logical units 2–4 are not handled by the LUN0 subprocess, but are passed on to the parent process via the "SUPER" mailbox. This mailbox has been sized so that it may buffer up to four complete data messages and three abort requests—the maximum number of valid transmissions that may be received from the VAL II controller while waiting for a response to a previous message. This

technique implicitly applies a lower priority to these messages compared to the LUN 1 messages.

Messages which are placed in the SUPER mailbox are read sequentially by the VAL II driver process using the queued I/O interrupt service. The interrupt handler uses the first three numeric fields of each message to update the state variables for the specified logical unit. If the message data field is blank and no further action is required, the VAL II message is acknowledged directly at the interrupt level. If further processing is required, the message data field is copied into a separate buffer reserved for the specified logical unit. Program control is then returned to the main routine which translates the data and writes the appropriate response to the LUN0 mailbox. This scheme relieves the main modules from the more mundane message handling, but also reduces the amount of time spent at the AST level. This is a necessary requirement if a balanced response time is to be provided to both the VAL II controller and the higher levels of the hierarchy.

As with the ORAC driver software, a permanent disk file is used to log the communications that take place between the VAX software and the VAL II controller. The VAX filename is generated by concatenating the character string "_VAL.log" to the name of the VAL II driver process. Both valid and invalid transmissions are recorded for all logical units except LUN 1. LUN 1 transmissions are not recorded due to the enormity of their numbers. Each of the four components of the VAL II message format are recorded in the log file in columnar format. The logical unit number is enclosed in square brackets and is followed by an arrow indicating the direction of travel.

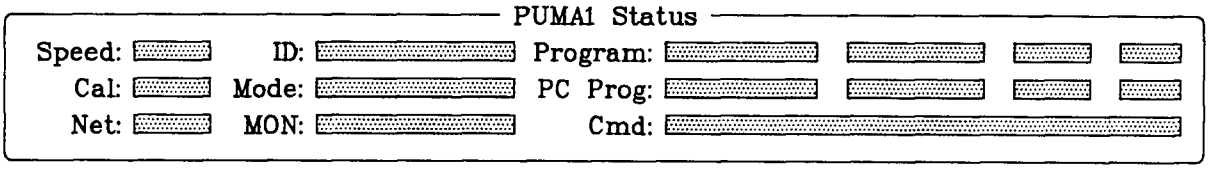

Figure 4.5: VAL II Driver Status Window

4.3.2 Status Display

The 8 state variables affected by the LUN 1 status messages represent a small subset of the approximately 65 state variables required to completely monitor the VAL II interface. From these 65, a subset of 15 variables has been selected for continual display through the SMG screen interface. The format of the status window used to display these 15 variables is illustrated in Figure 4.5. The name of the VAL II driver process which requests the window is included in the label when it is created. The actual placement of the window on the screen is left up to the SMG process.

The ID field is used to display a 13 character string which identifies the particular robot system being supervised. The string contains both the serial and model numbers of the robot, as well as the version number of the VAL II software currently executing on the controller. The Speed, Mode, and Calibration (abbreviated to Cal) fields combine to provide the current state of the robot subsystem. The speed which is displayed is the VAL II monitor *speed*, which is not a true rate, but rather a percentage scaling factor that is applied to all speeds specified within a program. The Calibration state variable is a simple on/off binary flag and is self-explanatory. The Mode variable indicates the

Table 4.3: Valid States for VAL II Subsystems

Robot Mode	Program Status	Command Monitor
ARM POWER OFF	RUNNING	NEXT COMMAND
MANUAL	PROMPT	READ CMD DATA
TEACH	WAIT	PROMPT
COMP	DRY.RUN	WAITING
PROGRAM	COMPLETED	WARNING
HOLD	PAUSED	ERROR
FATAL FAULT	HALTED	ABORTED
	ERROR	

current status of the robot's motion control software, and can take on one of the eight values listed in Table 4.3.

The current state of the VAL II program interpreter is displayed in the last four status fields of row 1 for Robot Control programs, and row 2 for Process Control (PC) programs. (Process Control programs are user-written status monitoring programs which may be executed in parallel with the actual robot motion programs.) The first two program status fields display the Name and Mode of either the current or most recently executed program. Of the eight Program Modes listed in Table 4.3, the first four are execution states which apply to currently executing programs; whereas the last four are completion states that indicate the reason why program execution terminated. The last two program status fields display additional completion information and are therefore cleared during program execution. If a program terminates due to an error condition, the error number corresponding to that condition is displayed in the third status field. The fourth field displays the line number of the last VAL II program instruction executed, regardless of the completion status.

The third row of the status window contains three fields which display information relating to the Supervisory communication network. The first field displays the number of network communication errors which have been detected. The Monitor field (abbreviated to MON) displays the current value of the combined state variable for LUN's 2 and 3. The seven valid states for this variable are also listed in Table 4.3. The Cmd field contains the first 34 characters of the last VAL II monitor command sent out over the network.

4.3.3 Start-up Sequence

The VAL II driver's first task is to assign a channel number to each of the mailboxes required for communicating with the cell management level. The Startup message is then sent to the parent process as confirmation that the VAL II driver subprocess is executing properly. A message is then sent to the SMG process requesting the creation of a new VAL II status window.

The next task is to create the I/O architecture required to communicate with the VAL II controller. The global page section is created and mapped to the common block containing the VAL II driver state variables, which are initialized to zero. The two mailboxes that will be used to talk to the network manager subprocess are created and assigned channel numbers along with a mailbox to receive a termination notice should the subprocess fail. The LUN0 subprocess is then created with the LUN0 and SUPER mailboxes designated as the default input and output devices. The default error device is set to point to a permanent disk file so that system errors may be trapped without interfering with the screen display. The file name is generated using the standard practice of concatenating the ".err" extension to the process name.

When the LUN0 subprocess has assigned all of the required I/O channels, and has successfully initiated the DDCMP protocol, it informs the VAL II driver process by setting its state variable to ACTIVE. The operator is then instructed to power up the VAL II controller and to enable the NETWORK and SUPERVISOR logical switches. These commands are required to initiate the Supervisor interface and must be issued at the VAL II system terminal. If the VAL II controller does not respond with a valid network message before a specified timeout, the VAL II driver sends an Error message to its parent process and terminates execution.

If a message is received, the initial state of the VAL II controller is determined by invoking the LUN 1 status monitoring function, and by issuing various status commands to the LUN 2 monitor task. If the robot is not calibrated, the operator is asked to make sure that it is in a safe location before the VAL II driver issues the calibrate command. The Ready message is then sent to the parent process, indicating that the VAL II driver is ready to accept commands.

4.3.4 Command Vocabulary

The VAL II driver software recognizes a command vocabulary made up of fifteen keywords with specific qualifiers and parameters as outlined in Table 4.4. This command structure is similar to that used by the ORAC driver module and the same rules and conventions described for Table 3.7 in Section 3.5.2.3 apply. With the exception of the SET_DIRECTORY command, the VAL II driver commands correspond directly to the equivalent VAL II monitor commands.

Table 4.4: VAL II Driver Command Syntax

```
CALIBRATE

SPEED  <value>

ENABLE   <VAL II switch>
DISABLE  <VAL II switch>

DELETE  [/ALL]          <program>   [,<program>,...]
        /PROGRAMS       <program>   [,<program>,...]
        /LOCATIONS      <location>  [,<location>,...]
        /REALS          <variable>  [,<variable>,...]

SET_DIRECTORY  <directory>

STORE  [/ALL]        [VAX]     [/COMMENTS]    <file>  <program>  [,<program>,...]
       /PROGRAMS     /FLOPPY   /NOCOMMENTS
       /LOCATIONS
       /REALS

LOAD   [/ALL]        [VAX]     [/COMMENTS]    <file>
       /PROGRAMS     /FLOPPY   /NOCOMMENTS
       /LOCATIONS
       /REALS

DO  <instruction>

EXECUTE     <program>
PCEXECUTE   <program>

PCABORT
PCEND
```

Although the VAL II driver performs an automatic calibration during the initialization sequence, the CALIBRATE command may be used to force a new robot calibration at any time. The potentiometer vs. encoder data required for calibration is generally loaded from the local floppy disk drive unit during the VAL II boot-up sequence. However, a defect in version 2.0B of the VAL II operating software causes this action to fail sporadically. As a guard against this anomaly, the VAL II driver software first attempts to overlay the calibration data file for the particular robot from the local floppy disk drive. If this file is not found on the diskette currently loaded, the operator will be asked to load the VAL II system diskette into the disk drive. The operator is also asked for confirmation of the robot position and mode prior to issuing the VAL II calibration command.

The ENABLE and DISABLE commands provide access to the eleven VAL II logical switches. As noted in Table 4.1, six of these switches directly affect the operation of the network interface and should be used with caution. In particular, the NETWORK and SUPERVISOR switches must remain enabled at all times, and should not be disabled by user programs. The INTERACTIVE and DISK.NET switches are utilized in both modes by the VAL II driver, and in general should not be changed. The REMOTE.PIN and MESSAGES switches are initially enabled by the VAL II driver, but should be selectively disabled so that the network does not become bogged down by verbose programs. The DRY.RUN switch provides a means of testing the logic of programs that have been written off-line without actually moving the robot.

The DELETE, STORE, and LOAD commands provide a means of manipulating the contents of the VAL II controller's memory space. With the exception of the command qualifiers, the syntax is similar to the equivalent VAL II commands. All three commands accept one of the four Data Type qualifiers; /ALL, /PROGRAMS, /LOCATIONS,

and /REALS. For the DELETE and STORE commands this qualifier performs the same function as the comparable command variations found in VAL II. The Data Type qualifier in the LOAD command is simply used to append a suitable file extension if none is specified. The STORE and LOAD commands also take a File Source qualifier that denote the device that the named file is to be loaded from, or stored to. This facility enables programs currently loaded on floppy disks to be uploaded to the VAX. The /NOCOMMENTS qualifier may be used to conserve VAL II memory space by stripping all comments when downloading a file from the VAX. The SET_DIRECTORY command may be used to specify a default directory for all VAX files.

The DO command executes the single VAL II program instruction specified in the calling sequence. This facility provides an alternative way to download locations or variables to the VAL II controller. It offers the advantage of not requiring a separate editing session to create data files; but has the disadvantage of being slower than the LOAD command for any more than two or three records.

The EXECUTE and PCEXECUTE commands oversee the execution of programs which have been previously loaded into the VAL II controller. Unlike the parallel VAL II commands, these VAL II driver commands do not return to the command monitor until program execution has terminated. Consequently, the multi-tasking utility of the VAL II PCEXECUTE command is lost, and it functions identical to the EXECUTE command. Multi-tasking can be achieved, however, by executing the PC program from within the Robot Control program by using the VAL II PCEXECUTE and PCEND program instructions. If this option is used, any currently executing PC programs should be terminated prior to sending the VAL II driver EXECUTE command. The PCABORT and PCEND functions have been added to the VAL II driver vocabulary for this purpose.

4.3.5 Special Considerations for Programs Using ALTER

The VAL II operating system supports real-time path control of robot motions through the ALTER function. This function modifies the programed motions by adding cartesian offsets to the current location in either a cumulative or non-cumulative mode. The required offset data can come from two different sources. In the EXTERNAL mode the offset data is transmitted to VAL II from an external computer over a dedicated serial line. Alternatively, the path control data can be computed by a Process Control program running in parallel with the Robot Control program. This latter mode is called INTERNAL ALTER and is the mode currently be used for force-control experiments in the UBC Mechanical Engineering Department.

The INTERNAL ALTER mode requires the Process Control program to send a complete set of offset data to the motion control software every 28 ms. If this mode is used in conjunction with the supervisory control interface the LSI 11/2 microprocessor is placed under a set of time constraints that it can't always meet. Because the ALTER mode is given a higher priority, the possibility exists that some network communications may be interrupted. The result is a lost of synchronization between the supervisory computer, which is still waiting for the message, and the VAL II controller, which believes that the message was sent. These conflicts can be avoided by adhering to the following programming guidelines:

- Remove all unnecessary TYPE and PROMPT statements; or disable the REMOTE.PIN switch so that these instructions display their data on the VAL II system terminal, rather than over the network.

- Always start the PC program first, and provide at least a 1 second delay between the PCEXECUTE and ALTER instructions.
- Always stop the ALTER mode and PC program explicitly. The NOALTER, PCEND, and STOP instructions should be executed in the stated order, with at least 1 second delay between each instruction.
- Use the REACTE instruction to specify an error handling subroutine that will perform the above shutdown sequence if an error occurs.
- Perform all time delays using the TIMER() function rather than the DELAY instruction. The VAL II interpreter implements the DELAY as a series of robot motion commands which specify the current location as the destination. This implementation produces unpredictable results at the best of times, but even more so when the ALTER mode is active.

4.3.6 Error Handling

The VAL II driver software is responsible for detecting and reporting communication errors and for relaying errors that are detected by the VAL II controller, in the same manner as the ORAC driver. The recovery options offered to the operator are dependant on the actual conditions but are generally the same as those offered by the ORAC driver, that is to skip or repeat the offending command.

VAL II user programs may also report application specific errors by using the Type instruction. The VAL II driver software checks the first 10 characters of all messages received from LUN 4 to see if they contain the "PGM ERROR" identifier. If present, the

VAL II driver treats the transmission as an error message and expects the corresponding error number and text to follow the identifier in that order. These values are stored in memory until the program terminates. If the program terminates through a VAL II STOP instruction these values are written to the program state variables and the operator is informed as if a VAL II error was detected. However, if a legitimate VAL II error occurs before the STOP instruction is executed it will cause the user supplied values to be overwritten.

Chapter 5

SUPERVISORY LEVEL SOFTWARE

5.1 ORCAM

The name ORCAM is an acronym for ORAC Computer Aided Manufacturing. It is a supervisory level software package that utilizes the ORAC and VAL II drivers to carry out the specific task of automatically machining a number of user-defined parts. This task involves the coordination of the loading and unloading activities performed by the PUMA robot, with the NC machining activities performed by the ORAC/Z8671 combination. The majority of these activities are sequential in nature, and very little opportunity exists for overlapping. Accordingly, ORCAM issues its fixed series of driver commands in a sequential manner; waiting for each command to successfully complete before processing the next one.

5.1.1 Input Data

The sequence of events for automatic manufacturing requires a knowledge of the physical dimensions of the part—both before and after machining—as well as the name and location of the NC programs. ORCAM has been designed to obtain this information

from the output of the ORACAP package. ORACAP is capable of generating ORAC NC code for parts to be externally machined on one or two ends, without the use of the tailstock [13]. The NC programs produced are in a form suitable for downloading to the ORAC lathe, and may optionally include the auxiliary I/O instructions necessary for the Z8671 tool changer interface.

In addition to the NC code, ORACAP also generates CAD and file content information which it stores in a somewhat scattered group of 5–9 different text files. As explained in reference [13], the naming convention used for these files may either be based on a Part Mode, or be user-defined. ORCAM requires that the Part Mode be used, and that all output files be stored in one directory. The direct inputs to ORCAM are therefore the part name, the file directory, and the number of parts to be machined, which are all entered interactively.

5.1.2 VAL II Programs and Locations

The VAL II programs that are executed by the ORCAM software take advantage of the flexible nature of the VAL II programming language to reduce the number of programs required, the number of robot locations that must be explicitly taught, and the amount of variable data that must be communicated between the VAX and the VAL II controller.

One such feature is the ability to move to a given location with any number of different tool definitions. The PUMA 560 robot is equipped with a standard Unimation pneumatic gripper that has a parallel stroke of 18 mm. To accommodate the ORAC's full range of barstock diameters with this small stroke, it was necessary to design a set of fingers

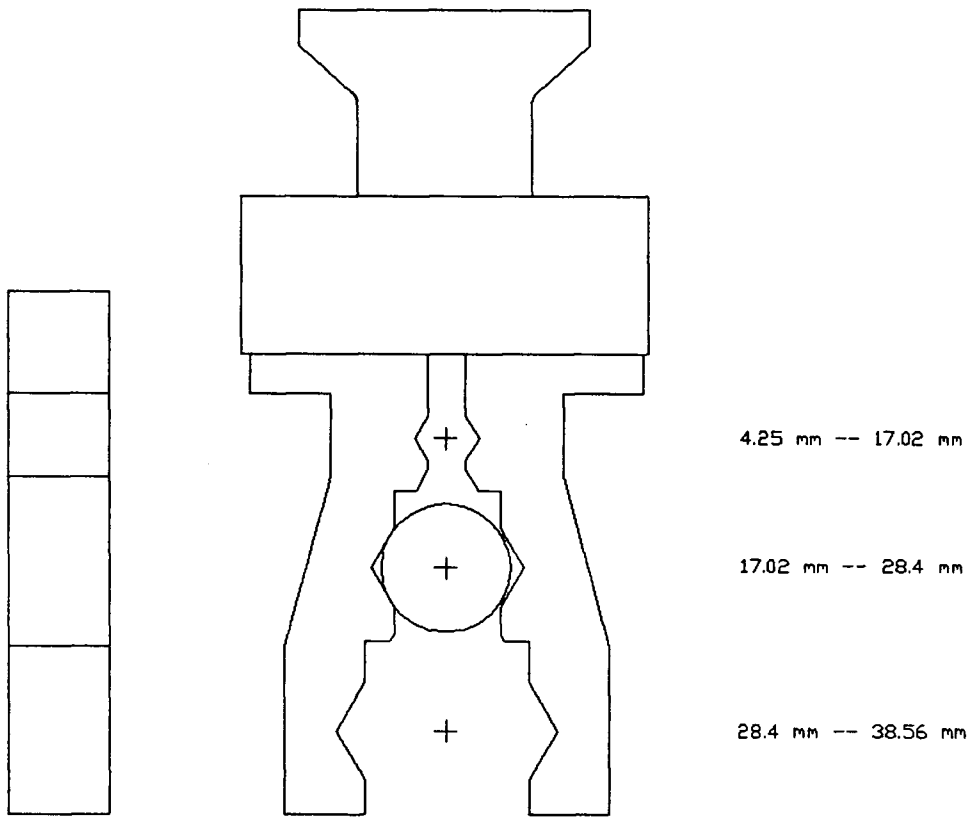

Figure 5.1: Gripper used to Grasp Workpieces of Varying Diameters

with three staggered grasping points, as shown in Figure 5.1. Each 120^0 V-notch centers its full range of workpiece diameters in the indicated position, and corresponds to one of three different tool definitions. These tool definitions are stored on the VAX and assume that both the Lord Corporation Remote Center of Compliance (RCC) and the JR^3 Inc. six-axis force/moment sensor are mounted between the robot's toolplate and the gripper. The correct tool definition is set by each individual VAL II program based on the diameter to be grasped, which is part of the data received from the ORCAM package.

The VAL II language allows robot locations to be defined in terms of cartesian transformations. These transformations may be explicitly taught by recording the current robot location, or they may be defined numerically in terms of both constants and variables. Transformations may also be combined to form new robot locations relative to existing transformations or to user defined coordinate frames. Frames are special transformations that are created by specifying the location of an origin and three other points that define the orientation and positive sense of the X and Y axes.

These features allow the VAL II programs to handle various workpieces without requiring a large number of taught points. The actual locations for each new workpiece are generated implicitly by applying relative transformations to pre-defined frames. Each relative transformation involves a single translation, and is generated numerically from axial offsets received from the ORCAM software. Because these offsets are transmitted as numeric constants rather than location variables, the amount of data that must be transferred for each new location is reduced by a factor of 12.

The locations for picking up the barstock are all defined relative to the input pallet. At the present time a temporary, single component, non-sensored input pallet is being used. It consists of a base plate with a 90^0 V-groove to cradle the barstock, and a back plate to ensure proper axial positioning. The position of the pallet is defined by a frame whose X axis is coincident with the root of the V-groove, and whose origin is located at the intersection of the X axis and the back plate. This frame is generated interactively by executing the VAL II program Teach.input. This program instructs the operator to place a prepared blank into the robot gripper, and to move the robot into position such that the blank is properly seated in the input pallet as shown in Figure 5.2. The current robot location is then recorded and the frame is generated from an accurate measurement

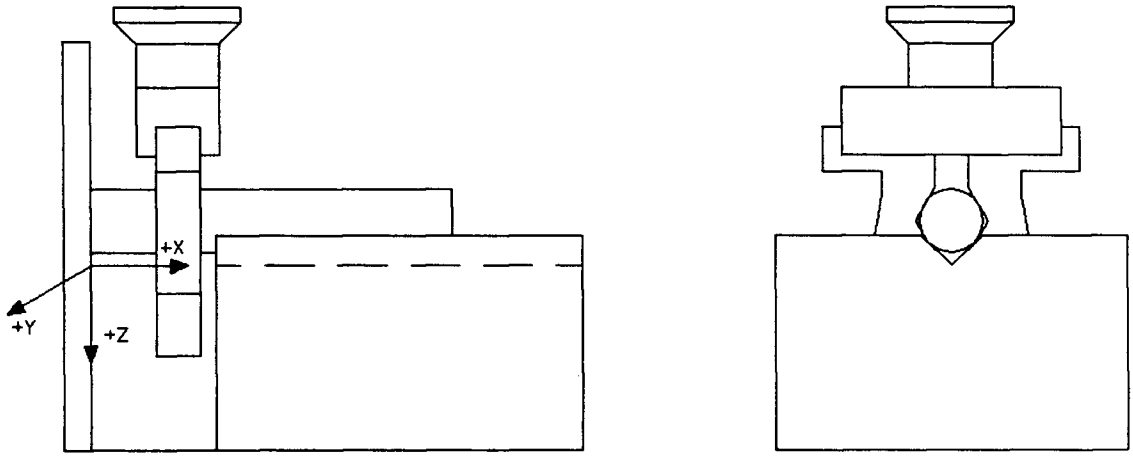

Figure 5.2: Reference Frame for the Input Pallet

of the barstock diameter and the axial distance between the back plate and the forward edge of the gripper fingers.

All of the locations required for loading the barstock, turning around the partially machined component (if required), and unloading the finished workpiece are defined relative to the lathe. The position of the lathe is defined by a frame whose X axis is coincident with the spindle axis, and whose origin is located at the intersection of the X axis and the outermost face of the chuck jaws. This frame is generated by the VAL II Teach.lathe program. This program instructs the operator to teach two locations that define the location of the spindle axis. As indicated in Figure 5.3, these locations are taught using a hardened steel pointer held in the robot's gripper, a specially designed chuck center, and a standard tailstock center.

At the present time, all finished workpieces are dropped off at the same location, so no frame is required for this operation. This Output location and a Safe home position

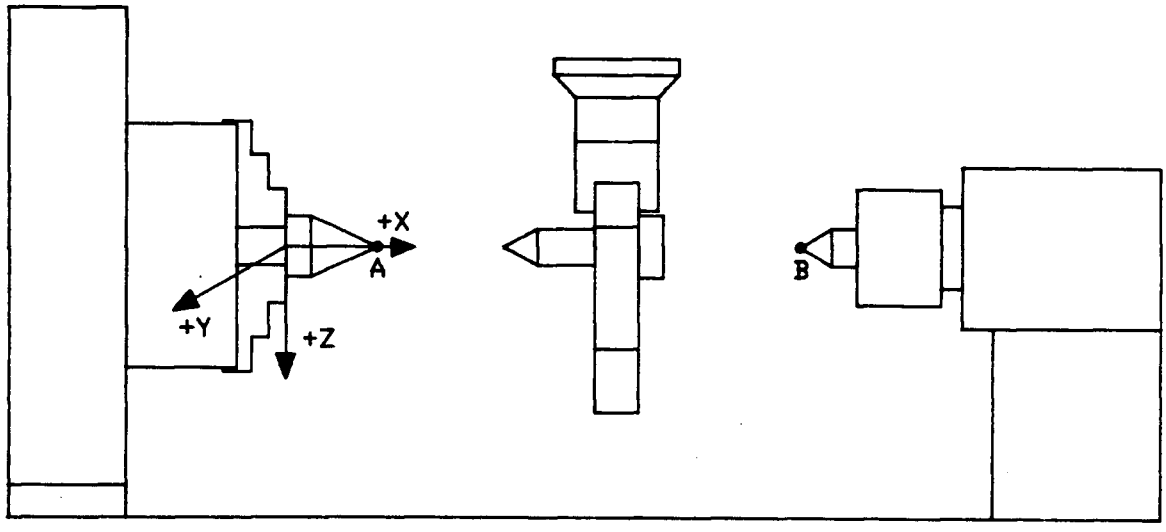

Figure 5.3: Reference Frame for the ORAC Lathe

are taught simply by moving the robot into the required position. Once taught, these two positions and the Input and Lathe frames completely define the workcell, and do not have to be retaught unless the equipment is moved with respect to the robot. Although the VAL II controller has battery backed memory, all locations are uploaded to the VAX as part of the interactive teaching process to guard against accidental loss.

The three major activities performed by the PUMA robot are controlled by three descriptively named VAL II programs; Load.chuck, Flip.workpiece, and Unload.chuck. Each program is responsible for both the generation of locations and the execution of a complete handling sequence; including the grasping, positioning, and releasing of the workpiece. To carry out this task, each program uses an appropriate subset of the nine input variables that completely define the workpiece, as outlined in Table 5.1. In each case the diameter variable is used to set the tool definition; the offset variables are used to define the position of the grasping or release point relative to the lathe frame; and the clamped.len variables are used to define the approach and retract positions required

Table 5.1: Workpiece Data Required by VAL II Programs

Load.Chuck	Flip.Workpiece	Unload.Chuck
load.diameter load.offset clamped.len[1]	flip.diameter flip.offset[1] flip.offset[2] clamped.len[1] clamped.len[2]	unload.diameter unload.offset clamped.len[2]

to clear the chuck jaws. The Load.chuck program also uses the indicated input data to generate the pickup location relative to the input pallet frame. The X axis offset is found by adding the Load.offset and Clamped.len[1] values, and the Z axis height adjustment is calculated from the Load.diameter value.

Each VAL II program is also responsible for the coordination of the robot motions with the opening and closing of the chuck jaws. Since the automatic chucking device has not yet been implemented, this coordination is presently done through a prompt to the operator, who must manually open or close the chuck jaws. While performing the requested action, care must be taken not to rotate the chuck as all VAL II programs monitor the Z8671 Spindle Running signal while the robot is in the lathe's work area. This signal is monitored on digital input 1003, and causes an interrupt if unexpectedly activated. The interrupt handling routine immediately stops the robot, opens the gripper, and then retreats out of the work area before generating a user detected error.

5.1.3 Calculation of Workpiece Data

The 7 diameter and offset values required by the three VAL II programs are calculated based on CAD data generated by ORACAP. This data includes the material, barstock diameter, the number of ends to be machined, as well as the length and shape of the machined section. The shape is defined in terms of a number of adjacent sections. Each section has a known diameter at both endpoints, and has a characteristic profile; circular, parabolic, linear, or threaded. Sections are connected at transition points that have a known axial position relative to a datum at the end of the workpiece.

The two Clamped.len values needed by the VAL II programs depend on the type of chuck jaws that are being used. The ORCAM package was originally designed to deal only with the *standard* jaws that are capable of gripping the ORAC's rated range of workpiece diameters. The ORCAM software has since been expanded to recognize the *oversize* jaws used for handling large barstock, but the workcell is still limited by the capabilities of the gripper, ie. 38.5 mm. Each set of jaws is characterized by a minimum and maximum diameter, a minimum clamping length, and a minimum offset to prevent interference between the jaws and the gripper body.

The first step taken by ORCAM is to check the barstock diameter against the minimum and maximum diameters for the gripper and the chuck jaws. If the diameter is in range, the variables corresponding to the chuck loading location are initialized. The load diameter is set equal to the required barstock diameter, while the load offset and first clamped length are set equal to the minimum values specified for the currently loaded chuck jaws.

The second task is to determine the length of the blanks required to machine the parts. For those parts that are to be machined on both ends, this is equal to the length of the machined section and is obtained directly from the CAD file. For those parts that are only to be machined on one end, a section of material must be added to allow the part to be clamped in the chuck. The amount added depends not only on the type of chuck jaws, but also on the shape of the finished workpiece. The list of CAD sections is scanned from top to bottom to find a location suitable for grasping the part after machining. This requires a linear section with both a constant diameter and a length greater than the width of the gripper fingers. If a suitable section is found, the added length is set equal to the minimum length required for clamping in the chuck jaws plus 5 mm to ensure that the cutting tool does not collide with the chuck. If the machined section doesn't contain a suitable unloading location, the 5 mm insurance section is replaced by the minimum length required to grasp the barstock without causing a collision between the gripper and the chuck jaws.

If the part is only to be machined on one end, all of the required data is now known. If a suitable section was found, the unload position is set to the first location in the section that has an offset greater than the minimum required by the chuck jaws. The unload diameter and offset are set accordingly. If no suitable section was found, the unload diameter and offset are set equal to the corresponding load values. In both cases the two Clamped.len variables are equated, and the "flip" variables are set to zero.

If a part is to be machined on both ends, the next task is to find a section suitable for clamping in the chuck once the part is turned around. Each section that is to be machined in the first pass is checked, starting at the outermost section. If the section isn't suitable for clamping, it is also checked to see if it will fit inside the 18 mm spindle

bore. If it does, the search continues; if it doesn't, an error is generated. If and when a suitable section is found, the second clamped length is set accordingly.

The flip diameter and the first flip offset are then set equal to the corresponding load values. This assignment is valid due to the fact that the ORACAP package always splits the part into two sections at a point located at least 40 mm from the end that is clamped first. The second flip offset is then calculated from the previous values.

The last step is to find a suitable unloading location for the parts to be machined on both ends. This position may be in any section that is far enough away from the chuck jaws, regardless if it is machined in the first pass or the second pass. If a suitable section is not found, an error is generated. If one is found, the appropriate unload values are set, and all nine values are then written to a data file in a form suitable for downloading to the VAL II controller.

5.1.4 Execution of ORCAM

The ORCAM task image may be executed directly using the VAX RUN command or it may be executed from the ORCAM.com command file. The command file option will ensure that all subprocesses created by the ORCAM software are properly deleted in the event of an unrecoverable error. The command file also sets the terminal type prior to execution, and should always be used when running the program on a terminal connected to the VAX through the terminal server.

The first task performed by the ORCAM image is the creation of the SMG process with its associated mailboxes. The operator is then asked if the PUMA robot is active. This option has been provided to allow the automatic features provided by the ORAC driver to be used even if the robot is not functioning properly. If the robot is active, the VAL II driver subprocess is created based on information contained in the ORCAM_machines.dat file located in the [kean.ORCAM] directory. This file contains the subprocess name, the driver task image name, and the terminal line to be used for both the VAL II controller and the ORAC/Z8671 interface.

When the VAL II driver has successfully completed its initialization sequence, the VAL II programs and the most recent locations are downloaded to the controller. The operator is then asked for confirmation prior to moving the robot to the required starting location. The operator is then given the opportunity to reteach each of the required frames and locations on an individual basis. If any of the locations are retaught, the current data is updated by issuing a STORE command to the VAL II driver.

The ORAC driver subprocess is then created using the name and terminal line specified for the ORAC/Z8671 interface. As with the VAL II driver, ORCAM waits for the full initialization sequence described in Chapter 3 to be completed before continuing. The operator is then prompted for the type of jaws that are currently loaded into the chuck. This completes the setup sequence, and the workcell is now ready to manufacture components.

ORCAM allows any number of different parts to be machined in varying batch sizes. The three required inputs of part name, file directory, and number of parts are prompted for at the beginning of each batch machining sequence. These inputs are then used to locate and translate the ORACAP CAD files. If any errors are found while calculating

the required offsets and diameters the operator is informed of the error and prompted for new input data. If the operator wishes to edit or regenerate the offending CAD files he must exit ORCAM. This may be done by entering a carriage return in response to the part name prompt. In this case the machine parameters are lost and the initialization sequence will have to be repeated when ORCAM is restarted.

If no errors are found while translating the CAD files, the resulting diameters and offsets are downloaded to the robot controller. The operator is then instructed to load the input pallet with the corresponding number, material, length, and diameter of blanks. (At the present time the input pallet has not been automated so each blank must be loaded individually.) As with any robotic handling operation, the initial placement of the blank affects the accuracy of the rest of the operation. Each blank must therefore be cut to length and faced prior to loading to ensure that it will sit square against the backing plate. Although the NC programs produced by ORACAP contain a facing pass, it is mainly for cosmetics and best results are obtained if the blank is faced to the proper length prior to loading.

When the operator confirms that the input pallet is properly loaded, ORCAM initiates the machining sequence by sending the appropriate SET_ZERO command to the ORAC driver subprocess. The blank is then loaded into the chuck by instructing the VAL II driver to execute the Load.chuck program. The first end of the workpiece is then machined by successively downloading and executing the necessary NC programs. For the majority of parts there will only be one NC program per end, but complicated parts using parabolic sections may require up to three different NC programs.

After the first end has been successully machined the workpiece will either be unloaded by executing the VAL II Unload.chuck program, or it will be turned around by executing

the Flip.workpiece program. If the part is turned around the second end is machined in a similar fashion, with the ORAC Z axis zero point being reset if necessary. This sequence is repeated until all of the requested parts have been machined.

The ORCAM package successfully demonstrates the communication abilities of the supervisory control package but it does not utilize the simultaneous processing capabilities. In fact the ORCAM package points out the major reason why the use of industrial robots as workpiece handling devices has not found widespread acceptance in plants with the traditional process-type layout. Because machining times tend to be much longer than handling times, the robot's capacity is not well exploited in the single machine application and is not economically justifiable [17]. The use of the manufacturing cell layout is a necessity if robots are to be justified in the machining industry.

5.2 FMC

The FMC cell management software was written to demonstrate the simultaneous processing capabilities of the supervisory control system. This program is an open-ended supervisory level package which may be used to directly access both the VAL II and ORAC drivers, and indeed any other drivers that may be implmented in the future. Unlike the ORCAM software which is dedicated to the task of machining parts, this package may be used to accomplish a number of different tasks, such as component assembly, or simple pick and place. The sequence of events required to accomplish the desired tasks must be decided externally and passed to the FMC package as text files.

Simultaneous execution of different tasks is supported in FMC by coding the instuctions for each task in separate files which are then processed in parallel. To facilitate this

parallel processing the FMC program keeps track of which machines are busy processing a command and which ones are idle. Idle machines are switched from one task file to another as the need arises. A *locking* mechanism is provided which may be used to group a number of commands into a single block. This ensures that critical operations which require the devoted attention of more than one machine are not interrupted. In particular, this mechanism should be used to ensure that each machine is in a suitable state before it is switched to another task.

Each individual machine is identified in the FMC environment by a user supplied tag which may be up to five characters long. These tags are read from the FMC_machines.dat data file which must be used to identify each new machine to the FMC program. In addition to the machine's ID tag, this file specifies the name of the executable image that contains the associated subprocess driver, and the name of the VAX terminal line used to communicate with the machine.

The syntax of the commands which may be included in an FMC input file are detailed in Table 5.2. The INITIALIZE command must be the first command in every FMC task file and must include the indentity tag of each machine accessed by the task. The LOCK command is used to gain exclusive access to the specified machines, and together with the UNLOCK command serve to group the intervening instructions into a single entity. The REPEAT and END_REPEAT commands function as a DO construct and may be used to repeat a group of instructions the specified number of times. The FMC program uses dynamic memory allocation to keep track of the return address and loop counter for each new REPEAT command so these constructs may be nested to any desired depth. Commands which are to be sent to an individual machine driver are preceeded by the identity tag associated with that machine.

Table 5.2: FMC Command Syntax

```
INITIALIZE   <machine ID>   [<machine ID> ...]

LOCK         <machine ID>   [<machine ID> ...]
UNLOCK       <machine ID>   [<machine ID> ...]

REPEAT   <number>
END_REPEAT

<machine ID>   <machine driver command>
```

The FMC program may be executed directly using the VAX RUN command or it may be executed from the FMC.com command file. The command file option allows the names of the FMC task files to be entered on the command line. If no names are entered, or if the RUN option was used, the FMC program will prompt for the file names interactively. In both cases, the first task file entered is the one that is processed first, and therefore decides the starting assignment for each machine.

Chapter 6

CONCLUSIONS AND RECOMMENDATIONS

The work described in this thesis has resulted in a functional FMC that is capable of producing turned components in a semi-automatic mode. The vast majority of the required operations including the generation, downloading, and execution of both NC and robot control programs have been completely automated. At the present time the operator is only required to load the input pallet and to operate the manual chucking device. This manual intervention is due to a lack of suitable equipment rather than any inherent limitations in the system. The present system is sufficient to demonstrate the principles of flexible manufacturing and has been invested with enough flexibility to allow the remaining two manual operations to be automated in the future.

The development of this workcell shows that stand alone machines may be successfully integrated into a supervisory control system. It has also demonstrated, however, that the level of integration achievable is highly dependant on the communication abilities built into these machines. The VAL II controller used to control the PUMA robot represents the high end of the spectrum and is capable of full integration. The ORAC lathe represents the lower end of the spectrum. Although many of the desired monitoring functions can be achieved by an external interface such as the Z8671 controller, the inability to directly sense the microprocessor's state is a limiting factor in these types

of machines. These limitations often make the time and money required to design the external interface unjustifiable in an industrial environment.

Although the machining capabilities provided by ORCAM represent the achievement of the original goal, the modular structure employed in the software design provide a sound basis for future expansion. The modular design and use of communication protocols allow a number of the components to be separated and put to other use. In particular the operator interface and the VAL II driver are completely general in nature and are not restricted to machining operations.

The FMC program has been designed mainly to support future development work. In order to take full advantage of its capabilities of simultaneous control more *machine* drivers are required. It is worth noting that the definition of a machine used by the FMC program does not restrict it to mechanical devices. Any device that exercises some form of control or monitoring function may be added to the supervisory control system, as long as it can communicate over a serial channel. One particular device which is seen as a possible target for future development is the JR^3 force sensor currently mounted on the PUMA robot. A communication interface between this sensor and the VAX 11/750 could be added to the FMC program to allow the development of an articulated assembly cell capable of changing the force-motion parameters of the robot to suit the desired task.

In addition to providing more device drivers for the existing FMC package, it is recommended that the FMC language be expanded to provide more functionality. While the current version allows simultaneous action of more than one machine, this feature is restricted to separate task files. In many instances this restriction does not provide enough connectivity between the operations of the individual machines. If complicated

tasks requiring a high degree of interaction are to be supervised, the FMC language should be expanded to provide:

- Parallel processing within the same task file.
- More structured constructs such as IF–THEN–ELSE.
- Instruction labels that may be used to refer to previous commands when specifying how long a machine should stay on a particular task.

Although the VAX 11/750 has proven to be satisfactory for the present work, its role as a multi-user system represents a limiting factor in its usefullness for an expanded control effort. If future expansion of either the ORCAM or FMC programs is to be undertaken it is recommended that they be transferred to the proposed VAXstation 3200.

REFERENCES

[1] A.P. Ambler, R.J. Popplestone, and K.G. Kempf. An experiment in the offline programming of robots. In *Proceedings of the 12th International Symposium on Industrial Robots*, Paris, 1982.

[2] M. Andrews. *Programming Microprocessor Interfaces for Control and Instrumentation.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1982.

[3] M.M. Barash. Towards the factory of the future. In *Computer Integrated Manufacturing Systems — ASME vol. PED-1.* American Society of Mechanical Engineers, November 1986.

[4] W. Barfield, S.L. Hwang, and T.C. Chang. Human-computer supervisory performance. In *Flexible Manufacturing Systems: Methods and Studies*, A. Kusiak, editor. Elsevier Science Publishing Co. Inc., New York, 1986.

[5] Ralph O. Buchal. *The Automatic Off-line Generation of Welding Robot Trajectories with Emphasis on Kinematic Feasibility and Collision Detection.* Master's thesis, University of British Columbia, September 1984.

[6] Ralph O. Buchal. *Computer Aided Programming of a CNC Lathe.* Ph. D. thesis, University of British Columbia, July 1987.

[7] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

[8] Denford Machine Tools Limited, Brighouse, West Yorkshire. *ORAC Programming Instruction and Maintenance Manual*, 1983.

[9] R.J. Grieve and G.W. Smith. Machine and component specification for flexible manufacturing systems for metal cutting processes. In *Flexible Manufacturing Systems: Methods and Studies*, A. Kusiak, editor. Elsevier Science Publishing Co. Inc., New York, N.Y., 1986.

[10] Mikell P. Groover. *Automation, Production Systems, and Computer Integrated Manufacturing.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1987.

[11] Yoram Koren. *Computer Control of Manufacturing Systems.* McGraw-Hill Book Company, New York, N.Y., 1983.

[12] E.B. Lambourne. Linkages required. In *Proceedings of the Plan For Computer Integrated Manufacturing.* Institution of Production Engineers, U.K., November 1984.

[13] Robert K. Moore. *Computer Aided Programming of a CNC Lathe.* Master's thesis, University of British Columbia, September 1985.

[14] Paul Ranky. *The Design and Operation of FMS.* IFS (Publications) Ltd., Kempston, Bedford, U.K., 1983.

[15] Kenneth L. Short. *Microprocessors and Programmed Logic.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

[16] Unimation Inc., Danbury, Connecticut. *PUMA 560 Programming Manual — User's Guide to VAL II,* 1985.

[17] H.J. Warnecke and R. Steinholper, editors. *Flexible Manufacturing Systems.* IFS (Publications) Ltd., Kempston, Bedford, U.K., 1985.

[18] Zilog Inc., Cupertino, California. *Z8 Microcomputer — Technical Manual,* 1983.

[19] Zilog Inc., Cupertino, California. *Z8671 Single-Chip BASIC Interpreter — BASIC/DEBUG Software Reference Manual,* 1983.