NISTIR 89-4046

THE WORKSTATION CONTROLLER OF THE CLEANING AND DEBURRING WORKSTATION

February 16, 1989

By:
Richard J. Norcross

National Institute of Standards and Technology
formerly
U.S. DEPARTMENT OF COMMERCE National Bureau of Standards Gaithersburg, Maryland

THE WORKSTATION CONTROLLER OF THE CLEANING AND DEBURRING WORKSTATION

Richard J Norcross

February 16, 1989

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

This chapter gives a brief description of this manual. The chapter begins with a short discussion of the requirements and features of the controller. Next the chapter describes the organization and the documentation conventions used throughout the manual. Finally the chapter concludes with a description of the manual's intended user.

1. THE CDWS CONTROL PROBLEM

The Cleaning and Deburring Workstation is designed to simultaneously perform multiple value-adding functions on workpieces from several manufacturing lots. Furthermore the workstation is to provide these functions economically, even on very small lot sizes. The workstation approaches the lot size problem by employing extensive on-line programming (or planning). The solution to the multiple-task problem comes from borrowing ideas from computer operating system theory.

2. ABOUT THIS MANUAL

This section introduces the organization of the manual and documentation conventions used.

2.1. Manual Organization

This manual is organized into seven chapters, four appendices, and a list of references. The seven chapters include three which provide the underlying theories for the controller and four chapters which outline some implementations of those theories. By understanding the basic theory, and reviewing instances of its implementation, a user will hopefully be able to use this approach in other applications.

Chapters one, two, and three give the background theory for the controller in increasing detail. This chapter, "Introduction", introduces the workstation controller and explains how to and who should use this manual. The second chapter, "System Overview", provides a brief description of where the controller fits into the AMRF hierarchy. It also contains an explanation of the basic approach used in developing the controller. Chapter three, "Architecture Description", outlines the data structures used in the controller. It concludes by describing how the various data structures interact to drive the controller.

Understanding how the workstation controller is run is beneficial to comprehending the underlying concepts. The fourth chapter, "Scenario of System Operations", explains the basic operations of the workstation. The chapter includes the execution of an example decomposition plan and explains solutions for predictive commands, program looping, and error recovery. Chapter five, "Interfaces to Other Systems", outlines the communication protocols to the equipment and cell level controllers, IMDAS, and the various background tasks performed at the host computer. The chapter includes the requirements on the interfaces for the controller's structure. The sixth chapter, "System Monitoring", provides methods used to monitor the activities and the state of the controller. Finally, chapter seven, "System Expansion", gives the location of the controller source code and

explains how to make additions and changes. Specific information is included on how to add new workpieces to the workstation's repertoire and how to add new on-line planning routines.

2.2. Documentation Conventions

This manual uses the following documentation conventions:

- In a control-key sequence, a caret (^) represents the CONTROL key. For example, control-C appears as ^C. Hold the CONTROL key while you press the C key.
- Information that appears on the terminal screen appears in bold courier print. For example, the **WSC>** prompt appears on the main terminal when the workstation controller system is awaiting input from the keyboard.
- Information that you must enter exactly as it appears in the manual is in plain courier print. For example, "**WSC>** `pause`" means to type `pause` and press RETURN after the WSC> prompt.
- Variable data that you must enter is represented within square brackets ([ ]). The instruction "**WSC>** `demo [part type]`" indicates that you are to replace `[part type]` with a specific block number when you enter the load command. For example, you might type `demo block_fh` and press RETURN after the WSC> prompt.
- Optional data that you may omit is represented within curly brackets ({ }). Most optional data is also variable data and appears in both curly and square brackets. The instruction "**WSC>** `RECEIVE_TRAY { [serial #] }`" indicates that you may either omit the serial # or specify a desired serial #. For example, the user might type `RECEIVE_TRAY` and press RETURN or you might type `RECEIVE_TRAY tray_123` and press RETURN.
- A modified BNF notation is used in data structure definitions. Specifically three periods, "...", indicate an optional repetition of the preceding entry. Italicized words in definitions indicate that the word is also a variable in the program.

3. INTENDED MANUAL USERS

This document is intended for NIST personnel and for government, industry, and university researchers interested in workstation control system implementations within the Automated Manufacturing Research Facility (AMRF). The manual assumes that the user is conducting research and developing applications for control of multiple robot systems, or implementing on-line programming and path-planning for those same systems. Although the manual includes basic instructions for commanding the Cleaning and Deburring Workstation, investigations beyond the

scope of this document would be required for fundamental changes to the system.

This manual assumes that the user is familiar with basic computer operating systems concepts. Additionally the user should be well versed in the lisp programming language, and in particular the Franz Lisp dialect. Understanding the Unix operating system would also be beneficial.

The specifics of the communication interfaces are not included in this manual. Users who intend to work on the interfaces should first read the specific reference for that interface. Scanning chapter five will provide the user with the name of the appropriate reference.

II. SYSTEM OVERVIEW

The Cleaning and Deburring Workstation's workstation controller (CDWS-WSC) is a second level (Workstation) controller in the AMRF hierarchy [1]. The controller accepts commands from and generates status for a "Cell level" external command source and a local terminal interface. Two Equipment level controllers accept commands from, and generate status for this controller. In addition, a two slot Tray Receiving Station, a Washer/Dryer unit, and a part holding vise indirectly take commands from the workstation controller. The controller has access to IMDAS (Integrated Manufacturing Data Administration System) [2] output via an Unix/Ethernet link to the data base's host computer (a VAX 11/780). The controller's connection to the AMRF network is via a Unix/Ethernet link to a Common Memory Manager on a separate SUN workstation. Major attributes of the CDWS-WSC are the control of multiple simultaneous tasks and implementation of on-line planning.

Unfortunately the AMRF's third level (Cell) controller is currently unable to provide multiple commands or respond appropriately to status messages. Therefore the Cleaning and Deburring Workstation has not been able to fully test these features. The current AMRF Cell level acts as an External Command Source which provides a fixed series of three commands for six possible tray configurations. For further information and directions on Cell level operations and expansions see reference [3].

The first of CDWS's two equipment controllers is an NIST/RCS system. This controller commands a PUMA 760 robot and associated pneumatic tools, sensors, and a wrist mounted quick/change device. Although the RCS controller accepts only single commands, the RCS system can accept new data while executing commands. However, the RCS data system has limited memory available for data and the workstation controller must track the memory usage and modify its contents as required.

The second equipment controller is a commercial VAL II controller from Unimation, Inc. The VAL controller directly controls a Unimate 2000 robot and indirectly controls the workstation's secondary equipment (tray transfer stations, washer, etc). Commands to this equipment pass through the VAL controller via Unimation's 32 line parallel I/O board to an RCS-type equipment controller. Communications with the VAL controller is an imitation of a human user typing on the controller's console. This communications requirement is the primary limiting factor in dealing with the VAL controller.

The Cleaning and Deburring Workstation is a data intensive system. Without the supporting data the workstation can perform no useful tasks. The primary repository for this data is the Integrated Manufacturing Data Administration System (IMDAS). IMDAS provides the workstation controller with tray contents and lot contents information along with copies of process plans and geometry specifications. IMDAS packages this information in a set of data reports [4]. The IMDAS writes these reports on the disk of a remote VAX 11/780. The CDWS-WSC copies these files via the Unix *rcp* facility to its local disk for reading. The CDWS-WSC also maintains local copies of all information for stand-alone testing.

The CDWS-WSC uses massive decomposition to simultaneously process multiple commands. The basis of this decomposition consists of three major module classes: job control blocks, resources,

and decomposition plans. A decomposition plan describes how to accomplish a given task and details the segmentation of the task. Furthermore decomposition plans can be created at run-time for immediate execution, thus implementing on-line planning. A job control block is roughly equivalent to a *process control block* in a computer operating system. A separate job control block maintains the status for each task, subtask, and subtasks of subtasks. The status of the job dictates the execution of the associated decomposition plan. Coordination between the various tasks is accomplished through resource allocation. A resource is any item, actor, or subroutine under the direct control of the workstation controller which allows only limited use (e.g., a robot executes only one command at a time). Each resource has associated with it scheduling and allocation functions, and information as to which task currently has access to the resource and which tasks are waiting. This combination of definition, status, and coordination permits the CDWS-WSC to efficiently control its various components in multiple cooperative and independent tasks.

Researchers have long attempted to improve the flexibility of automated systems by incorporating the ability to plan tasks in response to the changing environment. The CDWS-WSC implements on-line planning by interpreting decomposition plans at run time. Thus plans, based on the current environment, may be written for immediate execution by the controller. These plans may also include alternatives so that the system can respond to the more likely side events in a timely fashion.

III. ARCHITECTURE DESCRIPTION

The workstation controller is based on several basic computer operating systems theories. The controller uses process and precedence relationships to decompose multiple independent and coordinated tasks through an integrated system of critical sections and resource allocations. The system's communication and database requirements are thus not a direct function of the controller, but applications which the controller performs. This approach results in a very generic controller which is readily modified for new applications or changes in the old applications. The operation of the controller is implemented in the interaction between the three primary data structures used in the controller.

1. JOB CONTROL BLOCK

The job control block (JCB) is the central data structure of the system. The CDWS-WSC program uses a lisp record structure to form JCB's [5]. The structure contains twelve entries as shown in figure 1. Figure 1 also includes the default values and a specific description of each of the entries. Primarily, the JCB contains the current status of a task within the system. This status includes process relationships, execution data, and a list of allocated resources. Access functions for the job control block are in ~norcross/wsc/job_structure.l.

```
(defstruct (job (:default_pointer job)
          (:print "~a ~a ~{~a ~}"
                  (job-name job) (job-command job) (job-args job)))
                                ;   type   changeable?   accessed by?
          (job-name 'INTERNAL)  ;   atom      no             job
          (job-number 0)        ;   atom      no             job
          (job-command 'NOP)    ;   atom      yes            any
          (job-args nil)        ;   list      no             job
          (job-used_steps '(0)) ;   list      yes            any
          (job-comp_steps '(0)) ;   list      yes            any
          (job-offspring '(nil)) ;  list      yes            any
          (job-data nil )       ;  a-list     yes       job & offspring
          (job-resources ())    ;   list      yes            any
          (job-parent 'INTERNAL) ;  atom      no             job
          (job-level 0)         ;   atom      yes            any
          (job-status '(SETUP))) ;  1 atom    yes            any
```

Figure 1. Job Control Block Structure Definition

Each task has one parent and zero or more offspring. This relationship creates a process hierarchy in the form of a directed rooted tree. The data which identifies the process relationships consists of the JCB's parent, its offspring, and an identifying number. A task completes as its offspring

complete (a task may also force the offspring to complete prematurely). The number is the step name of the parent's task which generated the JCB's task. When a task completes, it modifies the parent's execution data to reflect the completion.

The execution data includes the steps completed, the steps currently executing, the command name, a status name, and a data list. The command name specifies the decomposition plan to use. The combination of steps completed, steps in execution, and status name determines the next activity in the execution of that plan. The data list is an associative linked list of variable names and values. Elements of the list are shared in hierarchal fashion along the decomposition hierarchy. Sharing the data provides communications between related tasks.

The primary method of task coordination is resource allocation. Allocated resources are the resource data of the JCB. A task may not execute until all of its required resources are allocated and none may be allocated unless all are available.

2. RESOURCES

The system allocates resources in a modified, by demand basis. A task may assume resources only when all of its desired resources are available. The availability of a resource is defined by that resource's check program. For most resources, the resource is available to a task when it is currently held by the task's parent or is not held by any task. If a resource is not available, the task is placed on a waiting list for that resource. When the resource becomes free, the tasks waiting for the resource return to the active queue in an order specified by a sorting function specified for that resource.

A record structure defines a resource as an object. As shown in figure 2, this structure contains six elements. The access functions for this structure are in ~norcross/wsc/resources.l.

```
(defstruct (resource (:default_pointer r)
                     (:print_name "~a" (resource-name resource)))
           (resource-name 'null_resource)
           (resource-user_list nil)
           (resource-wait_list nil)
           (resource-wake_list nil)
           (resource-sort_func '(lambda (x y) t))
           (resource-check_prog nil))
```

Figure 2. Resource Structure Definition

The user_list is a list of job control blocks. These JCB's are the hierarchy of ownership of the resource. As outlined in the first paragraph, resources are often inheritable. This permits a task to prevent interference from other tasks by holding a resource during it's decomposition.

The wait_list is also a list of JCBs. These tasks are suspended while waiting for the resource. Because there may be alternative sets of resources, a task may be on several wait lists. When a task assumes one set of resources it is removed from all other wait lists.

The wake_list provides for combinations of resources. For example the tray station consists of five tray sectors. Before a task can operate the tray station, all five sectors must be available. Thus if a task needs a tray station while one sector is busy then the task joins the tray station's wait_list, but would be included in the sorting when the tray sector becomes free. The wake_list is thus a list of resources. For more on the wake_list, see section VII on adding new resources.

The sort_func is a lambda definition used with Franz lisp's **sort** expr. The sort function requires two entries and returns **nil** if the order of the entries is incorrect, or non-nil if the order is correct. For example, since tasks are stacked onto the wait_list, the default lambda definition in figure 2 produces a LIFO order.

The check_prog is a single lisp statement which returns nil if the resource is not available or the user_list if it is available. When a resource is a composite or alternative resource, the check program effectively selects the appropriate resource for use. For further information on resources, composite resources, and alternative resources see section VII.

3. DECOMPOSITION PLAN FORMAT

A programmer describes the applications of the control system with decomposition plans. Although decomposition plans are based on process plans and state tables there is no method for translating either format directly into a decomposition plan. The plan's structure is a Lisp list structure containing either three or four elements. The first element is the resources required to execute the plan. The second element contains arguments to the plan and their default values. The bulk of the plan is the decomposition steps which are the third element. The final, and optional, element in the plan is a short character string which describes the function of the decomposition plan. The format of each of the elements is also defined in lisp and may not be clear to users unfamiliar with the basics of the lisp programming language.

The resource requirements are given as a list of lists of resources. Each list of resources is an alternative set of resources which the plan can use for decomposition. The order of the resource sets establishes the preference between the alternatives. Selecting which resources are needed for a particular plan is a more art then science. In the example in figure 3 the resource requirements are given as *C-760*, the PUMA 760 robot, and *VISE*, which is the part holding device. To teach the

```
(add-plan teach
          (((C-760 VISE))
           ((part-type nil) (sector 1) (tray 1) (instruction_set local))
           ((1 (0) ( (get-job-data 'grips) ) INSTRUCTION NOP ())
            (1 (0) ( (set-job-data 'grips (read-fixturings ($$ part_type))) )
               INSTRUCTION NOP ())
            (1 (0) () INSTRUCTION ABORT ( "No fixturing data" 100))
            (2 (1) ( (set-job-data 'part 'teach-part) ) MACRO add-part
               ( `(location ,(concat 'TRAY ($$ tray) ($$ sector))) )
            (3 (1)   . . .
```

Figure 3. Sample Decomposition Plan

part the other robot will also be required to move the workpiece, but it can be acquired after partial decomposition.

The plan's arguments are given as an associative list. Each element in the a-list consists of an argument name, as the key, and a default value. The system assigns variables by specified groups or by default order. When using default order the values of the arguments are supplied in the order given in the decomposition plan. The argument list's default values are used when the argument is omitted from the call and the value is not available for inheriting.

As mentioned above, the bulk of the decomposition plan is the decomposition steps. The steps specify how the plan is to be decomposed and in what order. Each step contains a step label, a list of precedence steps, a list of prerequisite functions, a decomposition descriptor, a command, and a list of arguments.

The step label is a number that identifies the step. An "or" relationship is established between multiple steps by assigning identical labels. Order in the decomposition plan establishes the precedence between like numbered steps. Step number zero "0" is executed when the decomposition starts and is not to be used as a label. In the example in figure 3 the first step is number "1" and there are three step number "1"'s. Therefore the first *step 1*, whose prerequisites are met, would execute while the others would be ignored.

The list of precedence steps is the first condition for executing a step. The order of the precedence steps is not important but they should be the minimum set. In figure 3 step 2 can be executed after step 1 and step 0 are completed. However since step 1 also requires step 0, the precedence list for step 2 need only include step 1.

The list of other functions is a catch-all in programming of the controller. The functions are executed sequentially until one of the functions returns "nil". If all the functions return non-nil the remainder of the step definition is set up as a new decomposition and the step label is added to the "used" list. The functions can check for proper data in the system, as in the first step 1 of figure 3, can modify system data, as in the second step 1 of figure 3, or perform any other purpose the programmer deems appropriate. The functions can be any lisp function but should be kept short as time consuming functions would clobber overall system performance. Likewise the functions should not contain input functions since lisp i/o functions hang the lisp environment until completed. Data for the functions should be taken from the job data list or the controller's global data base. Using global variables may cause unpredictable results when several decomposition plans are executing simultaneously.

The decomposition descriptor assists in setting up the next decomposition job. The system recognizes three descriptors. MACRO indicates that the command represents another decomposition plan. INSTRUCTION indicates that the command is a lisp function. Finally, "nil" or "()" indicates that no activity is required.

The command is an atom. This atom is the name of the lisp instruction or the name of another decomposition plan. Calling an unlisted command creates an error which automatically aborts the task.

The arguments element is a list of the parameters needed for the command. If the command is an

INSTRUCTION all of the arguments must be included in the argument list in the default order. However if the command is a MACRO only those arguments not already in the data list need to be included. For example in figure 3, the lisp function "ABORT" is a lisp lexpr with two arguments, a string for display on the terminal and a number for flagging the external command source. Meanwhile the decomposition plan definition for "add-part" has three arguments; part, part-type, and location. Since part and part-type are defined in the task prior to the call on add-part, only the location is needed in the call, and then only if the default would be incorrect. Since location is not the first argument in add-part, it must be specified when it is the only argument provided. Thus the argument list in step 2, when "eval"'ed once, produces the list "(location TRAY11)".

The final element in the decomposition plan is a string of sixty (60) or less printable characters. The string provides a very short description of the purpose of the decomposition plan. The string is displayed during execution time by a forgetful operator who can't remember the name or spelling of the command. In response to:

```
WSC> plans
```

the controller lists the name and message of all plans which have a fourth element. The operator then reviews this listing for an appropriate description.

A task terminates when its parent terminates or when it executes a **report** function. Absence of the report command hangs the task decomposition until the task's parent terminates. If the parent can not terminate without the task completing, the task would be suspended indefinitely.

4. WORLD MODEL

The controller plans many activities in response to the conditions in a locally held World Model. The world model is implemented as a set of global variables. These include the current workpieces, the locations in use, and the trays currently or soon to be resident in the workstation and the status of the memory on the RCS's RSL board. Programmers may include other data as required.

Three global variables; *locations, TRAYS,* and *places* , as defined in figure 4, contain the locations of the movable items in the workstation. These variables share their component elements (e.g., destructively changing the element in one variable changes the values of the others). The complete access functions for these variables are in ~norcross/wsc/locations.l.

Each workpiece has one entry in the a-list *locations* . This entry contains six elements. These include the workpiece's name, which is the key, the current location of the workpiece, its default tray location, the default tray sector, the workpiece type, and the workpiece's lot identification. The information in *places* and *TRAYS* is a summary of the location and tray elements.

The CDWS workstation controller supports the RCS database (RSL) by monitoring the amount of memory consumed by deburring files. When the total memory consumed exceeds 50,000 bytes the workstation controller directs the RSL memory board to create additional space by deleting some deburring files. The workstation also uses this data to determine if a new deburring file needs to be generated and/or downloaded to the RCS controller, and to track which loops need to be corrected by the deburring robot. Figure 5 gives the definitions of the global variables used in tracking the RSL memory board. The disk file /usr/local/norcross/wsc/RSLmemory.l contains the access functions.

```
locations := ( {[part location] ...} )
places    := ( {[location] ...} )
TRAYS     := ( {[tray] ...} )
part location := ( [name] [location] [tray] [tray sector]
                   [type] [lot id] )
name := string
location := ( [location name] . [location sector] )
location name := [ CDWS_TP1 | CDWS_TP2 | VISE | WASHER | ... ]
location sector := integer
tray := ( [tray id] . [tray station] )
tray id := string
tray station := [ CDWS_TP1 | CDWS_TP2 ]
tray sector := [ 1 | 2 | 3 | 4 | 5 ]
type := string
lot id := integer
```

Figure 4. Movable Item World Model Representation

```
RSL_memory     := ( {[RSL element] ...} )
part-count     := integer
teach-interval := integer
RSL element := ( [name] [Instruction Set] [RSL file time] [RSL file size]
                 [part count] [loops] [End Brush loops] [loops taught] )
Instruction Set := string
RSL file time := integer
RSL file size := integer
part count := integer
loops := ( {[loop label] ...} )
End Brush loops := ( {[loop label] ...} )
loops taught := ( {[loop label] ...} )
loop label := string
```

Figure 5. RSL Memory Usage Record

Each workpiece type has one element in the a-list *RSL_memory*. This element contains information on the deburring file and information on the teaching requirements. If the Instruction Set is different, or if the Instruction Set's file time is later then the RSL file time then the deburring pose file must be rebuilt. When the deburring file is generated the RSL file size entry is set to zero. Thus when that entry is zero, the deburring file must be downloaded to RCS. The "part count" entry is compared to global variable *part-count* to test if the loops on the part need to be retaught. Comparing the End Brush loops and loops taught determines which loops the workstation must command the deburring robot to reteach. For the fundamentals of robot teaching see reference [6].

5. REFERENCE DATA

The Cleaning and Deburring Workstation performs an extensive amount of on-line (or self-) programming. The data required for this task is maintained in the IMDAS system and on the local disk of the SUN computer.

Most IMDAS data eventually appears in the disk directory /usr/local/CDWS/db. The source of this data is either IMDAS, located on the CME VAX, or the directory /usr/local/CDWS/db_local. "db_local" contains a nearly complete copy of the IMDAS data used at the workstation.

The IMDAS data in "db_local" consists of Operation Sheets, Instruction Sets, Tray Contents Reports, and Lot Status Reports. The contents of the later two can be found in [4]. The Operation Sheets and Instruction Sets are process plans in the format of [7]. Actually there is only one Operation Sheet for the Cleaning and Deburring Workstation, *CDWS_CDWS_3*. This process plan is a data file which correlates workpiece types with Instruction Sets. The Instruction Sets are also used as data files which associate edges with deburring techniques and establish the nominal processing order. The actual process planning of the CDWS is performed at the workstation at run-time. Local process planning considers the current command set, the information in the Instruction Sets, the information in the World Model, and, to a large measure, opportunistic scheduling [8]. The fundamentals of the approach are given in subsequent sections of this manual and in reference [9].

Other IMDAS data includes the workpiece's Geometry Modeling System (GMS) representation [10]. Very little of this data exists. Therefore there are no copies kept in the "db_local' file and the workstation is programmed to proceed when the data is not available (a geometry file in a local format is required, this file may be generated from the GMS file or entered by hand). If the GMS data is available it is stored in a directory reserved for the specific workpiece type.

The directory /usr/local/parts/[workpiece type] contains most of the workpiece specific data. This includes the GMS file (~/GMS), the locally generated geometry file (~/geometry), the deburring pose file (~/RSL), the robot gripping transformations (~/robot.*), the vise deburring position transformations (~/viseD.*), the vise handling position transformations (~/vise.*), the tray position transformations (~/viseT.*), the part handling nodes (~/nodes), and the RCS conversion of the part handling nodes (~/oloc). The formats for these files are in reference [10], section VII.2.2.1.1, reference [11], section VII.2.2.2.2, section VII.2.2.3.2, section VII.2.2.3.2, section VII.2.2.3.2, section VII.2.2.4, and reference [11] respectively.

6. INTERACTION OF THE DATA STRUCTURES

The JCB represents tasks. Tasks may be either blocked, active, or pending. If a task is active it has complete access of the system to perform any function it requires (in the program the active task is the global variable *job*). A task may be blocked by either a resource request or by its offspring. When a task is blocked by a resource conflict, its JCB is placed on a waiting list for that resource. If a task is blocked by its offspring, the JCB floats, tracked by the system only by its process relationships. When a task is not blocked and not active, the task is pending and its JCB is on an Active Queue (the global variable *jobs*). The Active Queue is a simple linked list with new tasks added at the tail and newly-unblocked tasks added at the head.

When a JCB reaches the head of the Active Queue it is removed from the queue and is executed. If the task is a decomposition plan, the task is checked for resource allocation and further decomposition. Tasks may also be calls into the lisp executing environment. This procedure, of advancing the Active Queue and executing the active task, repeats until the commanded task is

completed. Additional commanded tasks may be added to the Active Queue at any time without interfering with the execution of other tasks.

The first effort of a newly created decomposition is to determine if the required resources are available. If one of the resources is unavailable then the JCB is added to the waiting list for that resource and the system returns to the Active Queue for another task. Once the resources are available, they are assigned to the JCB and the task is reviewed for decomposition. A review for decomposition is considered a critical section in the execution of the decomposition plan. The system checks each step of a decomposition plan before another task may become active. The system performs three checks on a decomposition step; is a step with that number not executing already, are all of the prerequisite steps completed, and are all of the prerequisite functions non-nil. JCB's are initiated for those steps which meet all three criteria. These JCB's are placed at the tail of the Active Queue and the procedure is repeated. If there are prerequisite steps still executing the task is blocked by its offspring and goes to sleep (dropped from the Active Queue). If a prerequisite function is nil, the JCB returns to the tail of the active queue so that the function can be tested again.

Eventually the task decomposes to a group of lisp function calls. These calls create Unix sub-processes, compute plans, modify system data, and stuff command mailboxes. The calls are always considered successful and always complete the task.

The system described is both simple and flexible. However several improvements must be made before the system performs to its potential. The longer term goal is to permit a very generic program representation so that the execution of the controller will be divorced from the programming of the controller.

IV. SCENARIO OF SYSTEM OPERATIONS

The Cleaning and Deburring Workstation's Workstation Controller provides an environment for the execution of control applications. Under this approach the controller does not specify how a task is accomplished. Instead the specifications are expressed in decomposition plans. This section reviews one of these decomposition plans and explains solutions to a few implementation specifics. The intention here is to show the techniques described in the previous section such that the source code may be understood and expanded.

1. *test*

test is a short decomposition plan used in system diagnostics. The decomposition plan utilizes several of the major components of the controller including parallelism, resource conflicts, and the hierarchal data structure. The following sections explain the decomposition plan's definition and review its execution.

1.1. The *test* Decomposition Plan

Figure 6 is the *test* decomposition plan. The format is the standard three element list form used throughout the source code. The three elements are resources, parameters, and body. The resource list contains one alternative, "(TEST)". Therefore, to execute, the task must possess the resource "TEST". The parameter list has one entry, "msg", which has a default value "nil". The body of the decomposition consists of four steps. The first two steps execute in parallel while the remainder execute sequentially.

```
(add_plan test
   (((TEST))
    ((msg nil))
    ((1 (0) (($$ msg) (set-job-data 'delay 10)) () NOP ())
     (1 (0) () MACRO test ("testing...1...2...3..."))
     (2 (0) ( ($$ msg) ) () NOP ())
     (2 (0) () MACRO test ("...3...2...1...test out"))
     (3 (1 2) (($$ delay)
               (set-job-data 'delay (+ ($$ delay) (sys:time))))
        printline (($$ msg)))
     (3 (1 2) ((set-job-data 'delay (+ 5 (sys:time)))) () NOP ())
     (4 (3) ((is_later ($$ delay))) INSTRUCTION report ()) )))
```

Figure 6. *test* Decomposition Plan

Both steps 1 and 2 check if the msg data is nil (*$$* is a macro which accesses the hierarchal data structure) and, if so, initiate new tasks. The second predicate function in the first alternative of step 1 also creates the data element *delay* which is used in step 3.

Step 3 is the *and join* of steps 1 and 2. And like steps 1 and 2, step 3 has alternative selections. If

the value of *delay* is non-nil then the value of delay in the hierarchal data structure is changed to the sum of the *delay* and the lisp function *sys:time* (sys:time returns the value of the Unix system clock). Then the system creates a task which calls the lisp function *printline* with the value of *msg* as its argument. If the value of delay is nil then *delay* is set to 5 seconds after the current time and no task is formed

Step 4 creates a busy wait. The lisp function *is_later* compares the Unix system clock to its argument value and returns nil if the argument is greater. Thus the task executing *test* would continuously return to the Active Queue until the current time is later then the value of the data element *delay*. Once the current time exceeds the argument value, the system generates a task for *report* which terminates the decomposition.

1.2. *test* Decomposition

In response to "**WSC>** test" the workstation controller (WSC) creates a task with the command *test* and with the argument list *nil*. For this example let us assume the task's job control block (JCB) is given the label "j0". Since *test* is the name of a decomposition plan the task is a MACRO.

The first step in the decomposition of a MACRO is to acquire resources. For the decomposition of *test* the resource "TEST" is required. Assuming this resource is available, TEST is assigned to j0 and the decomposition continues.

The second step of decomposition is parameter assignment. The *test* decomposition plan has one parameter, msg, which defaults to nil. Since j1's argument list is empty, j0 adds the data element msg with a value of nil to the head of its hierarchal data structure.

The first review of the decomposition steps generates JCBs (i.e., tasks) for steps 1 and 2. For both step 1 and step 2, the predicate function "($$ msg)" returns nil. Thus the second alternative of each step is executed. Since the execution of these steps does not immediately complete, the remainder of the decomposition body is blocked by offspring and task j0 is suspended. Meanwhile step 1 becomes task j1 with command *test* and the argument list ("testing...1...2...3...") and step 2 becomes task j2 with command *test* and argument list ("...3...2...1...test out").

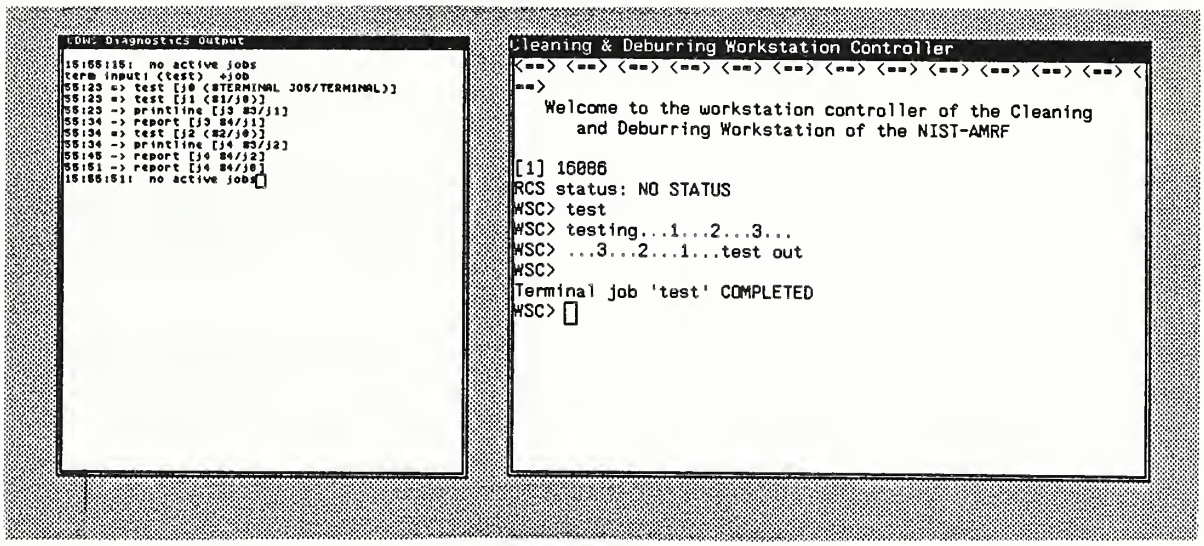
Since step 2 follows step 1 in the decomposition plan, j2 will follow j1 in execution. j1, as a MACRO, first checks for the resource TEST. Since the resource is held by its parent (j0), j1 assumes control of the resource. However when j2 attempts to acquire the resource, TEST is held by its sibling and cannot be acquired. Thus j2 is placed on the resource wait list for TEST while j1 continues execution.

Parameter assignment for j1 uses the default order method. The one element in the argument list, "testing...1...2...3...", is matched with the one parameter, msg. Since j1's data list is initially set to j0's data list, there already exists an entry for msg in j1's data list. Therefore that entry's value is changed to "testing...1...2...3...".

Since the data value for msg is non-nil, steps 1 and 2 execute the first alternative in j1. The predicate function in the first alternative of step 1 creates a new data element "delay" with a value of 10 and adds it to j1's data hierarchy. The commands in steps 1 and 2 complete immediately and step 3 is executed.

During the execution of j1, *delay* has a value of 10. Thus the first alternative of step 3 is selected. The value of delay is set to 10 seconds after the current system time and a task for printline (j3) is generated. Since j3 does not complete immediately, j1 is blocked by offspring and is removed from the active queue.

j3 is a task which executes a lisp instruction. *printline* prints its argument and a carriage return on the default stream as shown in figure 7b. Thus on its first review j3 executes completely and reports its completion to its parent (j1). As part of the report, j1 is added to the head of the Active Queue.



(a) (b)

Figure 7. Window Displays During *test* Decomposition
a) Diagnostics Window
b) Terminal Window

The next review of j1 finds steps 1, 2, and 3 executed and steps 1, 2, and 3 completed. Thus the system begins to execute step 4. For several passes the predicate function of step 4 will return nil. This causes the task to be returned to the tail of the Active Queue. During this delay the user can review the jobs list, the resource wait list, and the data structure to verify the performance of the system.

After 10 seconds the predicate function in step 4 returns non-nil and the system creates a task to perform the report function. The report function advises j0 that j1 has completed, returns j0 to the head of the Active Queue, and releases the resources held by j1. Releasing the TEST resource automatically returns j2 to the Active Queue.

When j0 is reviewed, steps 1 and 2 are executed and step 1 is completed. However since step 2 has not completed, the task is still blocked by offspring and is again suspended.

When j2 is reviewed the first effort is to acquire resources. Since j1 is complete, TEST is again held by j0, j2's parent. Therefore j2 may acquire the resource from its parent and continue execution. The parameter assignment and execution is the same for j2 as it was for j1. Thus when j2 completes, j0 is returned to the head of the Active Queue.

Now when j0 is reviewed, steps 1 and 2 are completed and the decomposition checks step 3. Since delay has no value for task j0, the second alternative for step 3 is selected. This alternative completes immediately and step 4 is reviewed. Step 4 again conducts a busy-wait. During this delay the operator can verify that the *delay* elements for j0, j1, and j2 are separate. When the delay finishes, the monitoring displays for the diagnostics and terminal windows should be as in figures 7a and 7b. (Note that j2 does not appear in the diagnostics window until the task has acquired the resources).

2. LOOPS

A useful programming construct which was not shown in the previous section is loops. Loops are a repeat of a number of decomposition steps. A loop is created with the function *restore.* Understanding how the restore command works, will assist the operator in understanding how the system maintains the status of a task.

In the previous section the status of a task was referenced by listing the steps which had been executed and the steps which had been completed. The job control block maintains two lists for this information. These lists consist of the labels of the executed steps and the completed steps.

When a decomposition step is executed, a task is created for the step's command. Also the label for that step is added to an "executed" (actually called "job-used_steps") steps list. When reviewing the steps for decomposition, the system bypasses all steps whose label is already a member of that list. Thus each step is executed only once even when multiple entries share the same label.

When a task completes, the report function adds its label (carried as "job-number") to the list of completed steps of the parent. This list (called "job-completed_steps") is then used in comparing the prerequisite steps for an execution alternative.

```
(add_plan g#0001
  (( ... )
   ( ... )
   ((1 (0) ... )
    (2 (1) ((x?)) ... )
    (2 (1) ((restore 1) nil) ... )
    (3 (2) ((null (y?))) () NOP ())
    (4 (3) () () NOP ())
    (4 (2) ... )
    (3 (2) ((restore 4) nil)  )))
```

Figure 8. Loop Construction Forms

Therefore to repeat a set of steps these lists must be modified. The restore command takes a list of

Therefore to repeat a set of steps these lists must be modified. The restore command takes a list of step labels as an argument and removes those labels from the executed and completed lists. In figure 8, step 2 demonstrates a *do until* loop, that is the decomposition will execute step 1 once then loop through step 1 until the predicate function, x?, is true. If the first alternative to step 2 is omitted the structure becomes an infinite loop. Similarly, steps 3 and 4 form a *while do* loop on the predicate y?. The restore function can also be implemented as a subtask but the loop forms are more complicated.

3. PREDICTIVE COMMANDS

Systems, such as the CDWS-WSC, which are based on the ordering of tasks tend to produce sequential operations or parallel sets of sequential operations. These sequences produce an undesirable cost when one subsystem needlessly waits for another subsystem. For example while the workpiece is positioned by one robot, the other robot could be changing its tooling or preparing the data. If the second robot's command is withheld until the first robot finishes, the workpiece waits in position during the second robot's preparation. This problem could be resolved by sending additional commands. However the communications lag time would still be evident. The CDWS addresses the possibly lost parallelism by issuing commands before their due, i.e., predictive commands.

Currently, implementations of predictive commands at the CDWS only deal with the VISE. The WSC does not issue commands directly to the VISE. The two robots, each of which may control the vise, have a communication link which establishes a *vise area lock-out*. Thus when the robot working in the vise area controls the lock out, issuing a vise related command to the other robot is safe. Therefore, when a robot has control of the lock out, it includes that information in its status message to the WSC.

Within the WSC the prediction feature exists on two levels of the decomposition. On the higher level, the instruction set directs the workpiece through sequences of deburring, washing, and buffing. On the lower level a deburring sequence consists of several positions in the vise and the deburring of associated edge groups. The current implementation uses resource allocation for the deburring sequence and communications through the data hierarchy for the upper level.

A deburring sequence is executed through the *DEBURR_EDGES* command. This command takes a list of position/deburring loop pairs as an argument and sequences through each by instructing one robot to place the workpiece at the position then instructing the Puma 760 to deburr the edge loops. To handle the predictive commands, DEBURR_EDGES acquires control of the VISE and passes control alternately to its two sub-goals (place and deburr). When, during the execution of the sub-goal, the movement or deburring task recieves status that the particular robot has control of the VISE, that task releases the VISE up to the DEBURR_EDGES task. Then the alternative sub-goal may acquire the resource, decompose, and issue its necessary commands. When there is no sub-goal waiting, DEBURR_EDGES modifies a data element which is the prediction mechanism for the higher level.

The parent of DEBURR_EDGES is the instruction set. The instruction set is a computer generated decomposition plan based on the process plan. The instruction set directs the workpiece through deburring, washing, and buffing operations and then returns the workpiece to the transfer station.

Since all of the instruction set's activities do not utilize the VISE, the VISE is not required for the decomposition. Instead the decomposition plan establishes a variable in the data hierarchy for coordinating the predictive commands.

The data element *predict* is placed on the data hierarchy by the instruction set. The decompositions of DEBURR_EDGES, WASH_PART, and BUFF_PART modify the value of *predict* during their final access of the VISE. The instruction set task, which remains active in a busy-wait throughout the decomposition, initiates the subsequent task when the value of *predict* changes.

This example shows two methods for implementing predictive commands; by manipulating resources and through the data hierarchy. Both approaches require extensive knowledge of the decomposition to be held in the decomposition plans of the subtasks. Hopefully future work will remove this requirement. Future efforts will also deal with developing predictive commands for a single actor.

4. ERRORS

There are several types of errors which may be encountered when operating the system. General error recovery is covered as part of the Operating Instructions in Appendix C. This section covers the theory employed in automatic error recovery. However, the implementations of these theories is currently very limited. Present implementations use operator intervention where the theory calls for sensors.

4.1. Error Recovery

There are three types of errors; those which are predictable but do not confuse the world model, those which are predictable and do confuse the world model, and those which are unpredictable. The world model is the data held in the controller and used to schedule and plan the activities of the system (e.g., workpiece location). Most instances of the world model becoming confused occur during the transition of an element of that data (e.g., workpiece movement). An error is unpredictable when a solution for that error has not yet been determined. The control system requires the ability to handle each type of error.

Errors which do not confuse the world model are handled in the decomposition plan. Since the effects of these errors by definition do not effect other tasks, the recovery is implicitly coded into the decomposition plan. Thus these errors are more alternative states than errors. Usually the solution is to restart the routine which failed.

The more serious situation is an error which confuses the world model. Normally these errors indicate that the world model is incorrect. The solution to these errors requires information such as what went wrong and what was the system trying to accomplish when the error occurred. In practice, sensors answer the first question while the decomposition hierarchy answers the second. The solution is to replace the decomposition plan where the world model data is relevant with another decomposition task which corrects the error. If the replacement task is unable to correct the problem the "prune and replace" procedure is repeated further up the decomposition.

A typical second type error is when a robot transfers a workpiece and finds its path blocked. The

supervisory controlier backs up the decomposition from monitoring robot status to where the movement decision was made. Sensors indicate that the robot is holding the workpiece and the robot is waiting near the delivery point. To recover the system attempts to clear the delivery point. If the next attempt to deliver the workpiece fails, the system backs up the decomposition to where the delivery point was selected and attempts to select a different delivery point. Once successful, the world model data is modified and the execution resumes. If the back tracking reaches the root of the decomposition tree then the error is considered unpredictable.

Unpredictable errors are ones which have no known recovery. Handling an unpredictable error consists of shutting down a minimal portion of the system and requesting operator intervention. A common unpredictable error is missing or incomplete data. Here the system expunges the task without any system reduction. The portion of the system which is shut down is defined by the resources held in the decomposition tree. System response to an unpredictable error is necessarily robust, which leads to laborious manual recoveries.

As a system matures, the solutions to many "unpredictable" errors will be determined and implemented into the system. An important aspect is that each "predictable" recovery is another decomposition plan.

4.2. Deadlock Detection and Recovery

Since the workstation controller schedules tasks based on resource allocation there exists the possibility of creating a deadlock condition. A deadlock exists when two or more tasks are waiting for resources which are held not by their parent, but by the parent of one of the other tasks. The controller has no mechanism installed for detecting or correcting deadlock. Since the workstation has very few common resources, a deadlock is an extremely rare occurrence. Actually the workstation has only one common inherited resource, the VISE, therefore deadlocks are impossible. The workstation avoids deadlocks through careful crafting of the decomposition plan. Therefore it is possible that a programmer may create a deadlock.

Deadlocks are detected by a lack of progress on a set of tasks. Recovery consists of terminating one of those tasks and restarting it once the other task has completed. Again this situation should never occur.

V. INTERFACES TO OTHER SYSTEMS

The CDWS-WSC communicates with five separate modules each with a unique protocol. These modules are the VAL II controller, the RCS controller, the AMRF data base, the External Command Source, and the various background functions provided by the workstation controller on the Sun computer. Each protocol is designed to meet the functional requirements of the module. At some future time all modules will be communicating through the common memory interface.

This section covers some of the fundamental aspects of each of the interfaces. Specific definitions are included in the appendices of this manual. Initialization procedures for the interfaces are in the Scenario section under "start-up". All interfaces can be reinitialized without a formal shutdown by the operator. That is, the initialization routines perform the required shutdown routines. Individual interfaces may be discontinued by setting flags as described in the Scenario section.

1. RCS INTERFACE

Communications with the RCS controller utilizes two RS-232 serial lines. The first port provides command/status communications and the second provides a path for RSL data. The protocols for the commands and the data are similar while a one byte protocol is used for status messages. The interface specifics are in Appendix E of this manual.

In summary the WSC-RCS command and data protocol transmits messages by lines. The receipt of the line is verified via an echo check. If the echo is correct, a two byte flag is sent indicating to the receiving station that the echoed data is valid. If the echo is not correct, the line is retransmitted. If the communications handshake is lost and the message becomes garbled the workstation controller directs the retransmission of the entire message automatically.

With the same protocol, the workstation controller relays data files to the RCS controller. RCS's data consists of robot poses computed from CAD geometry descriptions. At an appropriate time, the workstation controller creates a process which feeds the data to the RSL board of the RCS controller. Communication between the workstation controller and the sub-process is handled via the Internal Interface discussed later in this section.

The status of the RCS controller is represented in single characters. This approach permits the workstation controller to ignore the RCS status for awhile then catch up when its cycle permits. The available statuses are; READY, BUSY w/o VISE, BUSY w/ VISE, and various error conditions.

2. VAL INTERFACE

Unimation provides several ports for communications with its VAL controllers. However the controller at the CDWS has only one working port, the terminal port. Thus commands to the VAL II controller emulate a person typing on the VAL terminal keyboard. The commands consist of the defined monitor commands [12]. The status messages from the VAL controller are relayed from the terminal wire via a disk file to the workstation control program. The entire system is unfortunately

quite slow but has proven to be very reliable and easily supports communication error detection and recovery.

An intermediate process relays commands to the VAL controller. This process is created with one of the workstation controller's output ports as its standard input port. The process accepts one line, relays it to the VAL controller, and performs a character by character check on the echo. If the echo is incorrect, the line is retransmitted. The monitor function character, the first character on the line, is included in the echo check. Specifically, the VAL controller can only receive execution-mode commands when it is not executing, the workstation controller ensures the VAL controller is given only one execution command at a time and the monitor function character check in the intermediate process verifies that this limitation is not violated.

Nonexecution-mode commands (e.g., SIG, RLIST, etc.) can be sent during any phase of the controller. Therefore the monitor function character check is ignored. When commands for the VAL controller are preceded by the executing prompt ("*") the lead character check is not performed.

If a series of commands to VAL require interruption, the system kills the sub-process, creates a new sub-process, and transmits a "PANIC" command to the VAL controller. This procedure stops the execution of any program in the VAL controller. However the procedure also destroys all data without a record of what has already been sent. Therefore recovery must proceed from a point prior to any data transmission.

Data for the VAL programs is downloaded as a set of preliminary commands to the main command. These preliminary commands are in the "DO" form. For the assignment of real variables the command is:

```
DO [variable name] = [variable value]
```

Location and transformation data is sent via the "DO SET" variation such as:

```
DO SET [variable name] = TRANS([x], [y], [z], [o], [a], [t])
```

After all variables are set, the program is executed:

```
EXEC [program name]
```

The VAL status messages are handled by a separate process. This process reads all messages from the VAL controller, drops those which begin with four or more spaces (to avoid confusion between status messages and the command echoes; commands whose echo is not required by the controller are preceded by five spaces), and appends the others to a disk file on the Sun workstation. Using a disk file ("/usr/local/CDWS/data/val_messages") as the temporary repository of the status messages slows the entire process, but provides for non-blocking reading of the VAL status by the Franz Lisp environment.

The possibilities of errors in a terminal emulation is very high. However the echo checking routine, along with enforced limitations in the VAL program has made this protocol very reliable.

3. EXTERNAL COMMAND SOURCE INTERFACE

Communications with the level three controller (the external command source) and the Integrated Manufacturing Data Administration System (IMDAS) is via the AMRF Common Memory.

Connections to Common Memory are by means of a Common Memory Manager (CMM) residing on a remote SUN computer workstation. The CMM in turn provides communications network access via a Network Interface Program (NIP). Connections to the CMM are via Unix sockets between the SUN computer at the CDWS and the remote SUN computer.

The communications format with the external command source follows the AMRF communication protocol [13]. However the standard format presentation is modified so the controller works with a lisp style format. The transformation is done with the same function call which accesses the CMM and is invisible to the workstation controller. The lisp style format strips the redundant information off the common memory message and returns the data in a list structure. There is a similar lisp-style format for the status messages. Thus an appreciation of the communications protocols is not required by the workstation controller programmer.

The hierarchal command message format is a six element list as shown if figure (9). The first three elements are command number, update (or modification), and time. The fourth element is the transaction status. The fifth element is a list of order actions and the sixth element is always "nil". The transaction command list contains the transaction command, the transaction command number, and the nil element. The order actions list is an associative list based on the order action number. The elements in each associative sub-list is the order action update number, the command and a list of arguments or parameters. The argument list is also an associative list the parameter names as keys and one element which is the value of that parameter.

```
command message := ([cmd msg identifier] [cmd number] [cmd time stamp]
                    [transition cmd] [order actions]
                    [resource allocations])
cmd msg identifier := "CMDF"
transition cmd := ([transition number] [transition keyword]
                   [transition parameter list])
transition keyword := [ SYNC | WARM_STARTUP | WARM_SHUTDOWN ]
transition parameter list := nil
order actions := ([order action] {[order actions]})
order action := ([order number] [order update count] [action keyword]
                 [order command] [order parameters])
action keyword := [ EXEC | CANCEL ]
order command := [ RECEIVE_TRAY | SHIP_TRAY | DEBURR_LOT ]†
order parameters := ([order parameter] {[order parameters]})
order parameter := ([parameter name] [parameter value])
resource allocations := nil
"* time stamp" := ([yy] [dd] [hh] [mm] [ss] [mmm])
"* number" := [integer]
"* count" := [integer]
```

† any command for which there is a process plan could be given.

Figure 9. Command Message Working Format

The status format is a seven element list as shown in figure (10). This list contains an identifier ("FDBF"), a status message counter, a time stamp, an abbreviated echo of the command message, a transition status, and the actual status of commands. The echo of the command message consists of

the last command's number and its time stamp. The transition status is the transition command update count and a keyword taken from the AMRF Initialization Model [14]. A command status includes the order action's number, its update count, its status count, the task's status, the task's command, and some status parameters. These parameters are generally unique to the task and include a number code which identifies problems (ERROR_CONDITION), the number of workpieces in a multiple workpiece lot which are completed (COMPLETED), the number of parts scrapped (SCRAPPED), and a tray's intended pickup location (PICKUP_LOCATION). With the exception of the identifying numbers, the counters, and the order action's task status, the status message is ignored by the external command source.

```
status_message := (FDBF [status number] [status time stamp]
                        [command echo] [transition status]
                        [order status list] [resource requests])
command echo := ([command number] [command's time stamp])
transition status := ([transition number] [transition keyword]
                      [transition status parameter list])
transition keyword := [ READY | WARM_SHUTDOWN | COLD_SHUTDOWN ]
transition status parameter list := nil
order status list := ([order status] {[order status list]})
order status := ([order number] [order status count] [status keyword]
                 [order command] [order status parameters])
status keyword := [ ACKNOWLEDGE | BUSY | DONE | ERROR ]
order command := [ RECEIVE_TRAY | SHIP_TRAY | DEBURR_LOT ]
order status parameters := ([order status parameter]
                            {[order status parameters]})
order status parameter := ( [parameter name] [integer] )
parameter name := [ ERROR_CONDITION | COMPLETED | SCRAPPED |
                    PICK_UP_LOCATION ]
resource requests := nil
"* time stamp" := ( [yy] [dd] [hh] [mm] [ss] [mmm] )
"* number" := [integer]
"* count" := [integer]
```

Figure 10. Status Message Working Format

Commands from the External Command Source are subject to the restrictions outlined in the AMRF Initialization Model [14]. As such, commands from the External Command Source are ignored unless the current transition command is WARM_STARTUP and the current transition status is READY. Tasks which begin decomposition prior to a transition change continue execution until completed. However the status message for that command is no longer sent.

4. DATA BASE INTERFACE

The data base command mailbox is based on the ASN.1 standard format. Again a more convenient representation is used by the workstation and the conversion is invisible to the workstation operator. Unfortunately the data base status messages also have another format. The data delivered by the data base is received via disk files on a remote computer. The disk files are transferred to the

workstation using the UNIX *rcp* facility. The source code covering the IMDAS interface is in ~norcross/wsc/database.l.

Within the workstation the format of an IMDAS request is a list of two or three elements. The first element is an identifying integer which correlates to the IMDAS ASN.1 format and implements the AMRF Initialization Model for the IMDAS. The second element is a four character string which is the request identifier. The third, and optional, element is a DML string in an SQL format. Specialized decomposition plans implement the initialization protocol, the DML conversion, and the status interpretation.

The status message from IMDAS is also converted prior to use at the workstation. The status appears as an associative list with the identifier as the key and the request's status as the second element. The status is the ASCII value of the letters *A*, *B*, *D*, and *E* for acknowledge, busy, done, and error.

Specialized conversion routines translate the IMDAS reports into a useful form. The part, part type, and tray sector information in the TRAY_CONTENTS_REPORT becomes the World Model data identifying the locations of workpieces in the workstation. The PROCESS_PLAN/Operation Sheet becomes decomposition data which associates workpiece types with specific Instruction Sets. The LOT_STATUS report and the PROCESS_PLAN/Instruction Set bypass the actual controller and are used in on-line planning.

5. INTERNAL FUNCTIONS INTERFACE

The fifth set of protocols is between the workstation controller program and the other functions performed on the Sun workstation. Franz lisp supports the creation of various sub-processes with communication sockets. Unfortunately these become blocking I/O ports (i.e., a request to read one of these sockets will halt the execution of the program until something is read). Therefore communications with the background tasks is accomplished through a collection of disk files. These disk files provide flags which indicate when a process has completed or has encountered a significant error. Due to this flagging arrangement only one execution of the supporting task may occur at a time.

For example the Sun workstation provides the data management of the robot deburring paths. The function which transmits the pose file is called as a shell script with arguments. The arguments tell the program the desired part and the port to be used. When downloading is completed the program creates a disk file called **RSL.completed** in a flags directory. The controller program watches the disk for that file and continues the task decomposition when it is created. Similarly, if the downloading program has problems, it creates the disk file **RSL.error** and the control program initiates a recovery plan.

VI. SYSTEM MONITORING

The Cleaning and Deburring Workstation's workstation controller was not designed nor built for general usage. The monitoring facilities were designed for aiding development of the control environment not for monitoring the execution of tasks. Therefore the monitoring facilities are very primitive and may not aid the user at first. There are two types of facilities; a group of terminal display windows and a collection of query commands.

1. WINDOWS

There are three window displays during the execution of the workstation controller program. Each of the windows is a terminal emulator and uses the suntools *shelltool* application. Two of the windows; marked "CDWS Diagnostics" and "VAL MIMIC", cannot accept inputs. The third window, called "main terminal" and marked "CDWS Workstation Controller", accepts all terminal input to the system.

1.1. Main Terminal Window

By default the main terminal window appears in the left middle of the suntools screen. To use the terminal the mouse arrow must be in the main terminal window while typing. When initiated, the controller displays a welcome message in this window followed by a prompt. The wording of the prompt can be changed but always ends with a greater than symbol. For example:

```
WSC> set prompt Burrbusters
Burrbusters>
```

The main terminal window contains very general status information. Generally, three forms of messages appear at the main terminal; announcements of the completion of commands entered at the terminal, messages from subprocess of the controller, and programmed prompting for an operator.

The programmed prompting for an operator can be instructions or questions. The messages contain two or three parts respectively. The first portion is an answer key which consists of the letter "j" followed by a number in square brackets. The second portion is the message proper. The third portion, only for questions, is a guide to the appropriate responses. Answer a query by beginning the input with the answer key. For example:

```
WSC> [j23] Clear parts from Vise.
WSC> j23
WSC> [j12] Save New Process Plan? [n/number]
WSC> j12 n
```

1.2. Diagnostics Window

The diagnostic window appears along the right side of the suntools screen. The messages there generally describe the decomposition of tasks within the controller. Other, more specific, messages may be commanded from within decomposition plans. The decomposition messages are preceded by a time stamp.

The Diagnostics Window indicates the current activity of the controller. When there is no activity the Diagnostics Window shows a "**no active jobs**" message. This message includes a three figure time stamp (hour:minute:second). Other messages use a two number time stamp (hour:minute).

The decomposition messages use a simple shorthand to indicate controller activity. The message includes a type flag, the name of the command, the job number, and the parent's job number. For example in

```
14:20 => process-part [j1/j12]
```

the time stamp shows the message was created at 2:20 pm. The symbol "**=>**" is the type flag and indicates the controller is decomposing a plan ("**->**" indicates the decomposition is a lisp instruction). "**process-part**" is the name of the plan being decomposed. In the brackets, "**j1**" and "**j12**" are the job name and the parent's job name respectively.

The diagnostics window may be closed without effecting the performance of the controller. However, since the information in the window is very useful, the shelltool should not be terminated.

1.3. VAL Monitoring Window

The third window begins as an icon along the top of the suntools screen. The icon is a drawing of the Unimate 2000 with the word "mimic" beneath it. This window contains a monitor of the echoes sent by the VAL II controller. The window is useful when attempting to recover from errors involving the Unimate 2000. For most applications the window should be left closed.

1.4. Console Window

A fourth window is produced by the initial call to suntools and may be of interest. In the lower right corner is a terminal window with a reduce-sized font. Errors generated by the Unix operating system appear in this window. This window should remain open since operating system errors are generally significant.

2. QUERY COMMANDS

Table 1 contains commands which show the running status of the workstation controller. All commands are Lisp functions which print to the main terminal window.

Since the system rejects improperly formatted commands "syntax" is generally the most useful query. For example:

```
WSC> syntax teach
(part-type nil) (sector 1) (tray 1) (location nil)
WSC>
```

This response indicates that the teach command requires a part-type argument and the remainder of the arguments are optional providing the part is at (**CDWS_TP1 . 1**) (tray #1 and sector #1). When the list begins with a "nil" default, that argument is generally required. However, when the

"nil" argument follows "non-nil" arguments then the system can generally derive the information from the other arguments.

COMMAND	DISPLAYS
WSC> sup_command	current External Command Source (ECS) command
WSC> status	last status sent to the ECS
WSC> external_jobs	list of tasks initiated by ECS or at the terminal
WSC> jobs	list of jobs in the active queue
WSC> parts	lists parts in the workstation and their locations
WSC> plans	a list of decomposition plans and their purpose
WSC> pp_plan [plan]	pretty prints the decomposition plan definition
WSC> resources	lists the resources and their owner lists
WSC> STAT	combines "resources" and "parts"
WSC> syntax [command]	gives the input format for that command

Table 1 Common Query Commands

VII. SYSTEM MODIFICATION

One of the primary purposes for this manual is to assist subsequent operators in maintaining and expanding the system. This section contains instructions for three types of expansions. The first section explains how to improve the speed of the system and where to find source files. The second section outlines the procedure used in adding a new workpiece (or modifying an existing workpiece) to the system's repertoire. The third section explains the method used to implement on-line planning in the system. These three areas represent the most likely areas for immediate system expansion.

1. COMPILING CHANGES

The file "~/wsc/compile.l" contains a set of lisp "include" statements. Each file to be included in the base system should be listed in this file. The base lisp environment may then be created in two ways. For faster run-time performance the code should be compiled. For simpler access the program can be interpreted without machine optimization.

To compile lisp code, use the standard Franz Lisp compiler, **liszt** [5], with no options. liszt compiles each expr, lexpr, and fexpr, expands macros, replaces tail recursions, and performs other similar functions which provide for faster executions. liszt also provides the programmer with indications of bad code by identifying unintentional global variables. However, while non-working code will not work when compiled, working code may fail when compiled. Most of these problems stem from the use of special variables and cascading macros, which can be avoided.

The proper sequence for creating a compiled lisp environment is:

```
cdws-1> liszt compile.l
{ several pages of messages }

cdws-2> lisp
=> ?fast
=> ?ld cstart
{ several pages of messages }
=> (exit)
cdws-3> wsc.new
WSC>
{ test as desired }
WSC> exit
cdws-4> mv wsc.new wsc
cdws-5>
```

To work on the workstation without first compiling the code:

```
cdws-1> lisp
=> ?ld start
=> (run)
WSC>
```

An important note is that the information on the workpieces is data, can not be interpreted by the Lisp reader, and is therefore not listed in the compile.l file.

2. PROCESSING NEW WORKPIECE TYPES

New part types are added to the Cleaning and Deburring system via the "master" command. The "master" command functions as a workstation controller command and as a Unix shell command. However the Unix system recognizes the "master" command only when the user is logged onto the Sun computer as "rjn". The workstation controller executes only one instance of the "master" command at a time. Other requests are queued and processed in an unspecified order.

2.1. The "master" Command

The command "master" takes one to four arguments. The first, and required, argument is the part type. Each part type in the system must have a unique name. Using an existing name is immediately obvious as the existing data is displayed. Aborting and restarting "master" corrects the problem. The remaining command arguments are the desired instruction set name, and the part's tray and tray sector positions. These arguments default to "local", "1", and "1" respectively. If an operator intends to reuse the instruction set, he should not use the default instruction set name, since the data would soon be over-written.

The "master" function produces or modifies a large set of data. The data includes a geometry description of the part, a directory of workpiece gripping poses, and an instruction set. Currently IMDAS (Integrated Manufacturing Data Administration System, i.e., the data base) supports only the instruction set. To add the instruction set to the data base, copy the instruction set file to the VAX and contact the group currently responsible for maintaining IMDAS (contact the AMRF project manager for names and telephone numbers).

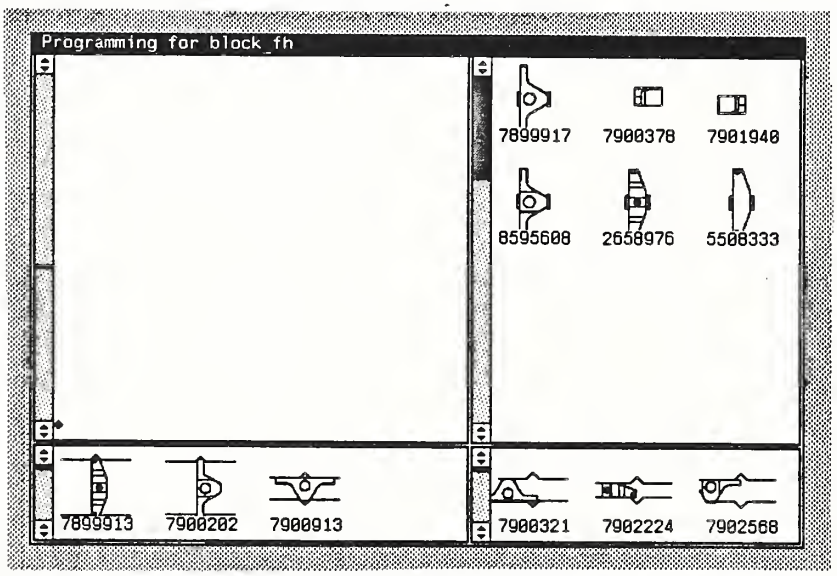


Figure 11. The master Interface Window

2.2. "master" Program Inputs

The "master" command initiates a SUNview window on the SUN computer station as shown in figure 11. The window has four sub-windows which display: the robot grip positions, the intermediate vise positions, the deburring vise positions, and the connectivity matrix (clockwise from upper right). The right mouse button activates a pop-up menu when the mouse is in the robot grips position sub-window (note: when first used, the menu may not appear until after the button is released). Inputs to the "master" program enter via this menu. The following sub-sections, although not in the proper menu order, discuss the effect of each menu command.

2.2.1. Add Geometry

"Add geometry" initiates two suntool windows. On the right side of the Sun's screen is a texteditor [15] window containing the part's geometry file. The left side of the screen contains a gfxtool [15] which shows the primary views of the part in the drawing portion and the mouse's position and the name of the selected edge in the text portion.

2.2.1.1. The Texteditor Window

Texteditor is an ASCII file editing utility tool provided by Sun Microsystems. A detailed description of the texteditor's features is found in Section (1) of the Sun Commands Reference Manual (reference [15]). As a minimum, the operator must be able to find, change, and save text.

The geometry file itself is in a simple Lisp readable format which does not require topological closure nor completeness. The first line of the file is **(/EDGES)** and the last is **(/END_EDGES)**. The intervening lists, marked by matched parenthesis, represent a single edge on the part. An edge is a portion of the intersection of two surfaces and is defined as per figure 12.

The surface definition for a plane contains the unit vector normal to the surface of the plane and the dot product of a point on the plane and the normal vector. The optional vector entry is not used for planar surfaces. The surface definition for a cylinder contains the unit vector of the axis of the cylinder, the radius of the cylinder, and a point along the axis of the cylinder in that order.

The order of the edge data may be significant. While for LINE curves there is no specific order for edge's end points, an ARC's end points are counter-clockwise about the axis of the cylindrical surface. The order of the surface coincides with the order in which the two surfaces were machined. The machining order determines the orientation of the machining burr. Thus the burr lies along the second surface and protrudes over the first machined surface.

```
edge := (key  point1  point2  surface1  surface2  sense  curve)
key     := string
point   := vector
surface := ( type vector  number  {vector} )
vector  := ( numberX  numberY  numberZ  )
type    := [ CYLINDER | PLANE ]
sense   := [ INSIDE | OUTSIDE ]
curve   := [ LINE | ARC ]
```

Figure 12. Edge Listing Definition

2.2.1.2. The gfxtool Window

The "add geometry" feature also uses a suntool "gfxtool". The gfxtool is a two panel window which contains a drawing in the lower panel and text in the upper window. A detailed description of the gfxtool is available in the SUN Commands Reference Manual section (1).

The graphics panel of the gfxtool displays a simple rendition of the workpiece as defined in the geometry file. This rendition is a "backedge elimination" drawing of the workpiece. Backedge elimination is a derivative of the more common "backface elimination". For each edge in the geometry file, the surfaces of the edge are compared to the view angle and if that surface *might* be visible, the edge is drawn. Planes might be visible if the normal vector is between -90° and 90° off of the viewing vector. The drawing routine translates cylinders into multiple planar facets when determining their visibility. This approach produces an x-ray view of the workpiece which may be confusing. However the operator can simply remove edges which should be blocked by other surfaces.

The left side of the gfxtool contains eight buttons. The lower six of these buttons select the view of the workpiece. The selections are the six primary unit cartesian vectors; (0 0 1), (0 0 -1), (0 1 0), (0 -1 0), (1 0 0), and (-1 0 0). In the upper left are two additional buttons; Draw and Quit. The Draw button causes the routine to reread and redraw the geometry file. Thus changes to the geometry file can be quickly reviewed. The Quit button terminates the program.

The gfxtool display is used in geometry preparation to review the geometry data. When an edge appears to be incorrect in the drawing the operator highlights that edge by selecting with the mouse device. The name of the edge then appears in the text panel of the gfxtool. This name is the entry into the texteditor. The operator may then make the appropriate changes to the geometry file, redraw the figure, and view the alteration. This procedure continues until the representation of the workpiece is correct.

2.2.2. Add Robot Grips

The **Add Robot Grips** menu command initiates a graphic interface for specifying how the robot's gripper can grasp the workpiece. The input to this interface generates a gripping pose in the part's coordinate frame and is independent of the robot. After the grip positions are specified, the workstation controller determines which grip position is appropriate for the immediate task. The connectivity matrix, discussed in a subsequent section, provides a quick summary of the coverage of the gripping positions.

2.2.2.1 Robot Grip Selection Window Display

The robot grip selection is a SUNview program with three drawing canvases. The large upper canvas shows a backedge elimination drawing of the part as viewed from the robot's wrist. The two lower canvases provide backedge elimination drawings as viewed from the right and left gripper pads. All three canvases contain cross-hairs. In the upper canvas the cross-hairs indicate the center of the gripper. In the lower canvases the cross-hairs show the center-bottom of the gripper pads.

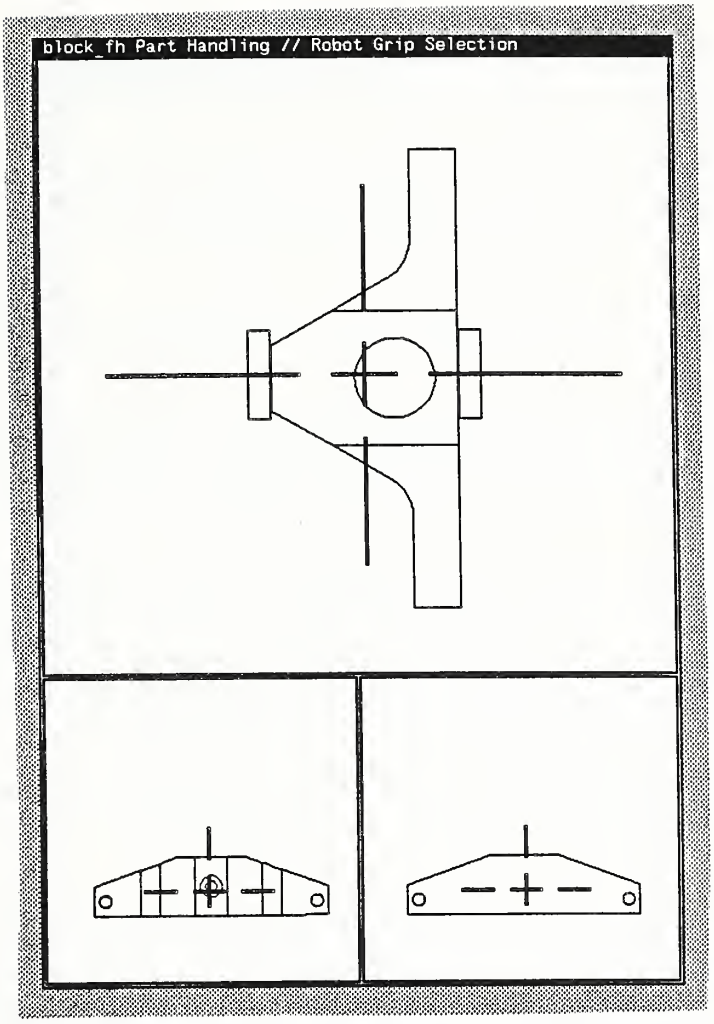


Figure 13. Robot Grip Selection Window

Within these windows, the left mouse button moves the center of the cross-hairs to the current mouse position. Thus the combination of mouse location and left button adjusts the position of the workpiece in the X-Y direction in the upper window and in the Y-Z direction in the lower windows. Inputs for workpiece rotation are via the menu interface.

2.2.2.2. Menu Inputs

The most inputs to the graphic interface are by means of pop-up menus. The pop-up menu in the upper canvas contains eight entries while the menus in the smaller canvases contain three each. The smaller menus are a subset of the larger menu and control only the workpiece location. The main menu also includes entries regarding the rendition and for saving the input.

The first element in the main menu is for saving the grip position. The system saves the grip position in the part's coordinate frame. This includes a 3x3 rotation matrix and a 3x1 position vector. Additionally the system saves the expected gripper opening and a shot-down drawing of the

part in the gripper. The information is saved in the workpiece's information directory as /usr/local/parts/[workpiece type]/robot.# where # is a unique integer.

The second menu element, "Set Approach Vector", is a means to directly set the robot's approach vector. The element initiates a confirmer window which permits the user to directly modify the 3x1 approach vector. Since the system automatically normalizes this input, a unit vector is not mandatory. Through proper use of the "Rotate" elements this menu element can be totally avoided.

The third menu element displays the workpiece's current transformation. "View Orientation" brings up an unalterable confirmer window which contains the 3x3 rotation matrix and the 3x1 position vector.

The system displays the workpiece as a backedge elimination by default. However the workpiece may also be drawn as a wireframe. The fourth menu element, "View Type", contains a pull-right menu which selects between the two renditions. This change effects all three display canvases.

Both robots at the CDWS utilize NIST designed split rail grippers. These grippers have significant operating range and are self-centering. That is, the gripper forces the part to the center of the gripper. The fifth menu item, "Close Gripper", simulates this action by centering the workpiece in the horizontal direction on the upper canvas's cross-hairs. The program accomplishes this by moving the pixels of the drawing. As such this is fairly slow and subject to misinterpretation of the pixels.

The sixth menu item, "Increment Center", has a pull-right menu matrix. This menu redraws the workpiece the specified number of pixels over in the specified direction. If the gripper has already been closed, this entry recloses the gripper on the repositioned part.

As a short cut to entering the approach vector by hand as described in the above paragraph, the seventh menu element, "Rotate", permits rotation about the current approach vector. Since this menu element is also available in the lower canvases, virtually any orientation may be realized through a combination of rotates on the orthogonal views. The "Rotate" element has a pull-right menu which selects the amount of clockwise rotation in uneven degree increments from 90° to -90°.

The eighth and final menu element,"Quit", terminates the robot grip select program. There is no confirmation used in the quit element. If the robot select is accidentally quit, it may be restarted from the "master" window.

2.2.3. Add Vise Grips

The **Add Vise Grips** menu selection initiates a suntool which assists development of gripping positions for the rotary vise. The vise is a self-centering fixturing unit used to hold workpieces during deburring and for part reorientation. The same suntool also helps the user set the default tray loading positions.

2.2.3.1. Window Display

The **Add Vise Grips** window contains a single drawing canvas as per figure 14. This canvas contains a backedge elimination drawing of the workpiece, central cross-hairs, and the leading edges of the vise jaws (when the vise jaws are closed).

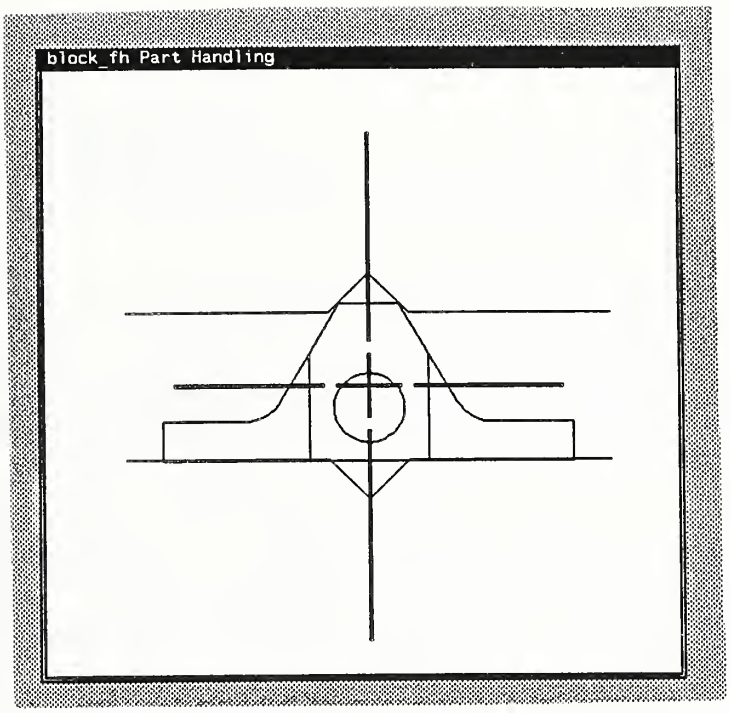


Figure 14. Vise Grip Selection Interface Window

The workpiece rests on a flat surface while the vise closes. Therefore the Z axis position is always taken from the lowest point on the workpiece (opposite the viewing or Normal vector) and there is no selection for the Z position. A warning message appears in the original shelltool if there are fewer then three vertex points at the extreme low distance.

As with the robot canvases, the vise canvas allows the position of the workpiece to be changed by selecting the left mouse button. Likewise there is a pop-up menu associated with the right mouse button. All commands to the vise grip selector are initiated by the menu.

2.2.3.2. Menu Inputs

The majority of inputs to the graphic interface are by means of a cascading pop-up menu. The menu contains nine entries which affect workpiece location, its visual rendition, saving the input, and terminating the session.

The first element in the main menu is for saving the grip position. The system saves the grip position in the part's coordinate frame. This includes a 3x3 rotation matrix and a 3x1 position vector. Additionally the system saves the expected gripper opening and a shot-down drawing of the part in the gripper. The system identifies vise grips as either a deburring position, an intermediate position, or as a tray position. Deburring positions represent sturdier vise grips for when force is applied to the workpiece. Intermediate positions are less sturdy and are used for reorienting the workpiece. The tray position is the default position for the workpiece in the transfer tray. The information is saved in the part's information directory under the title **vise*.#** where * indicates the

type (Deburring, intermediate, or Tray) and # is a unique integer.

The second menu element, "Set Approach Vector", is a means to directly set the robot's approach vector. The element initiates a confirmer window which permits the user to directly modify the 3x1 approach vector. Since the system automatically normalizes this input, a unit vector is not mandatory. Through proper use of the "Rotate" and "Flip" elements this menu element can be unnecessary.

The third menu element displays the workpiece's current transformation. "View Orientation" brings up an unalterable confirmer window which contains the 3x3 rotation matrix and the 3x1 position vector.

The system displays the workpiece as a backedge elimination by default. However the workpiece may also be drawn as a wireframe. The fourth menu element, "View Type", contains a pull-right menu which selects between the two renditions. This change effects all three display canvases.

The fixturing device at the CDWS utilizes a NIST-designed split rail gripper. The gripper has significant operating range and is self-centering. That is, the gripper forces the part to the center of the gripper. The fifth menu item, "Close Gripper", simulates this action by centering the workpiece in the horizontal direction on the upper canvas's cross-hairs. The program accomplishes this by moving the pixels of the drawing. As such, this is fairly slow and susceptible to misinterpretation of the pixels. For tray positions, the "Close Gripper" command orientates the workpiece in the tray guide (an L-shaped device used for positioning workpieces in the trays).

The sixth menu item, "Increment Center", has a pull-right menu matrix. This menu redraws the workpiece the specified number of pixels over in the specified direction. If the gripper has already been closed, this entry recloses the gripper on the repositioned part.

As a shortcut to entering the approach vector by hand, the seventh and eighth menu elements, "Rotate" and "Flip", permit rotations about the current approach vector and the horizontal vector respectively. Virtually any orientation may be realized through a combination of rotates and flips. These elements have pull-right menus which select the amount of clockwise rotation in uneven degree increments from 90° to -90°.

The ninth and final menu element,"Quit", terminates the vise grip select program. There is no confirmation used in the quit element. If the vise select is accidentally quit, it may be restarted from the "master" window.

2.2.4. Compute Nodes

Movement planning is done at run time via nodes. A node is a combination of a robot grip and a vise grip. These combinations are tested for each robot prior to run time. The program tests for collisions between the grippers and against each robot's joint limit to determine if the combination is possible. Nodes which share a robot grip or a VISE grip define a legal movement. Combinations of legal movements are the possible robot/VISE paths.

At computation, nodes are displayed in the upper left master window as a n×n connectivity matrix, where n is the number of permissible nodes (figure 15). The entries in the connectivity matrix are

the number of movements required to move the workpiece from one node to another. If the motion requires more then seven movements the matrix displays the "+" symbol. If the motion is not possible the matrix displays the "." symbol. Presence of either the "+" or "." symbol indicates that additional gripping poses may be required.

Although part handling by the Puma 760 is not implemented at this time, nodes are computed for the Puma/VISE combination. When a permissible node for one robot is not permissible for the other robot the connectivity matrix displays a "-" symbol on the main diagonal.

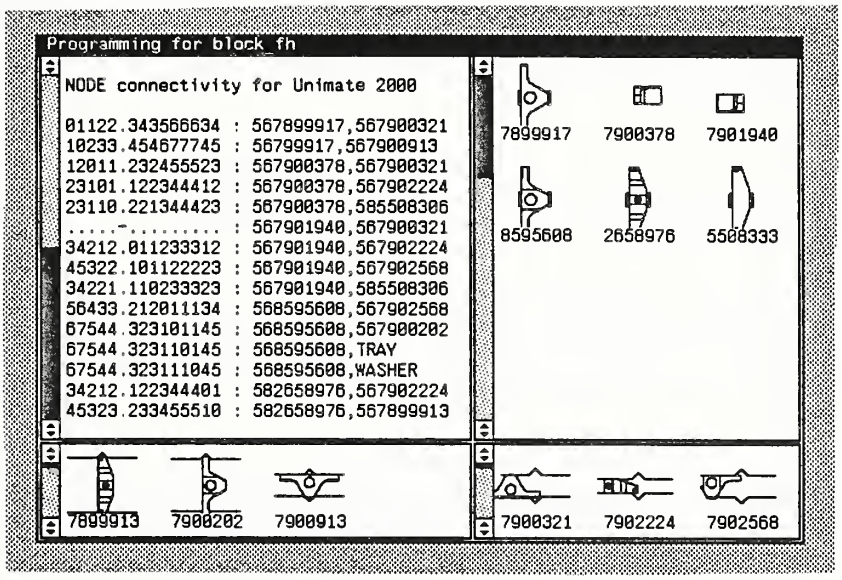


Figure 15. Node Connectivity Display

2.2.5. Redraw Icons

When fixturing positions are added their icons are not automatically added to the display. "Redraw Icons" causes the icons in the upper right and both lower master windows to be redrawn with the new positions included and the removed position omitted.

2.2.6. Edit Pose Directory

Poses are saved in the workpiece's information directory. Occasionally duplicate and unnecessary files must be removed. The "Edit Pose Directory" command opens a shelltool at the information directory. Standard UNIX shell commands may then be used to remove or modify the files. The most useful shell commands are *ls*, *more*, and *rm*. *ls* creates a listing of the files in the directory. Specifically "`ls r* v*`" creates a listing of the gripping positions.

All files in the information directory consist only of printable ASCII characters. The Unix

command *more* creates a listing of the file on the shelltool.

Most excursions into the information directory are to remove unwanted files. The *rm* command removes specified files (e.g. `rm r*234` removes the robot grip file whose identifying number ends with 234). Nodes must be recomputed after files are removed or a movement may be planned for which there is insufficient data.

2.2.7. Select Deburring

The "Select Deburring" menu selection initiates a window tool which allows the user to associate edges with deburring techniques and to establish the order of activities. Unlike the robot grip and vise grip window tools, the deburring selection tool does not use menus. This tool uses a single window which has four regions; a drawing canvas, view selection, deburring technique selection, and an ordering selection section.

The drawing canvas is the largest section of the window. Displayed on the canvas is a backedge elimination drawing of the workpiece. Clicking the left mouse button on a drawn edge selects, or de-selects, that edge for deburring.

Selected edges are highlighted with either a dashed or dotted line. The dashed line indicates that the burr is orientated out of the window screen while a dotted line indicates a less favorable orientation. The orientation of the drawing, and thus the relative orientations of the burrs, is chosen in the view selection section.

On the left side of the window is the view selection section. This section consists of the list of identifying integers of the deburring positions in the vise (see section VII.2.2.3 above). One integer in this list is highlighted with an arrow. The highlighted entry is the representation on the drawing canvas. A new view is selected by clicking the left mouse button over the desired entry.

Across the bottom of the window is the deburring technique selection. This selection displays a tool name and several adjustable parameters for that tool. There are currently three tools; an end brush, a hole brush, and a chamfering tool. Selection of the tool is round robin style by clicking the left mouse button over the tool name. The tool parameters include force: the limiting force used to program the robot; speed: rotational speed of the tool; rate: the speed of the tool across the workpiece; depth: the distance into the workpiece the tool is to move; and offnormal: the angle between the desired approach and the surface of the workpiece. Each parameter has a slide bar for adjusting the parameter's value along with the current value and the parameter's units.

In the upper left corner are six mouse buttons which make up the ordering selection. These buttons are divided into two groups; QUIT, DRAW, and SAVE, and BUFF, BREAK, and WASH. The SAVE button causes the selected edges, view (i.e., fixturing position), and deburring technique to be saved as one association. Each association will become a step in the process plan. The QUIT buttons cause the deburring planner to translate the saved associations into an Instruction Set type process plan and to terminate the program. The DRAW command clears the saved associations so the process plan can be reinitiated. Since the workstation's buffing wheels are not operational, the BUFF button causes the same activities as the BREAK button. The BREAK button causes a command which forces the order between the technique/view/edges associations. The on-line path generator attempts to combine associations such as to minimize the number of workpiece

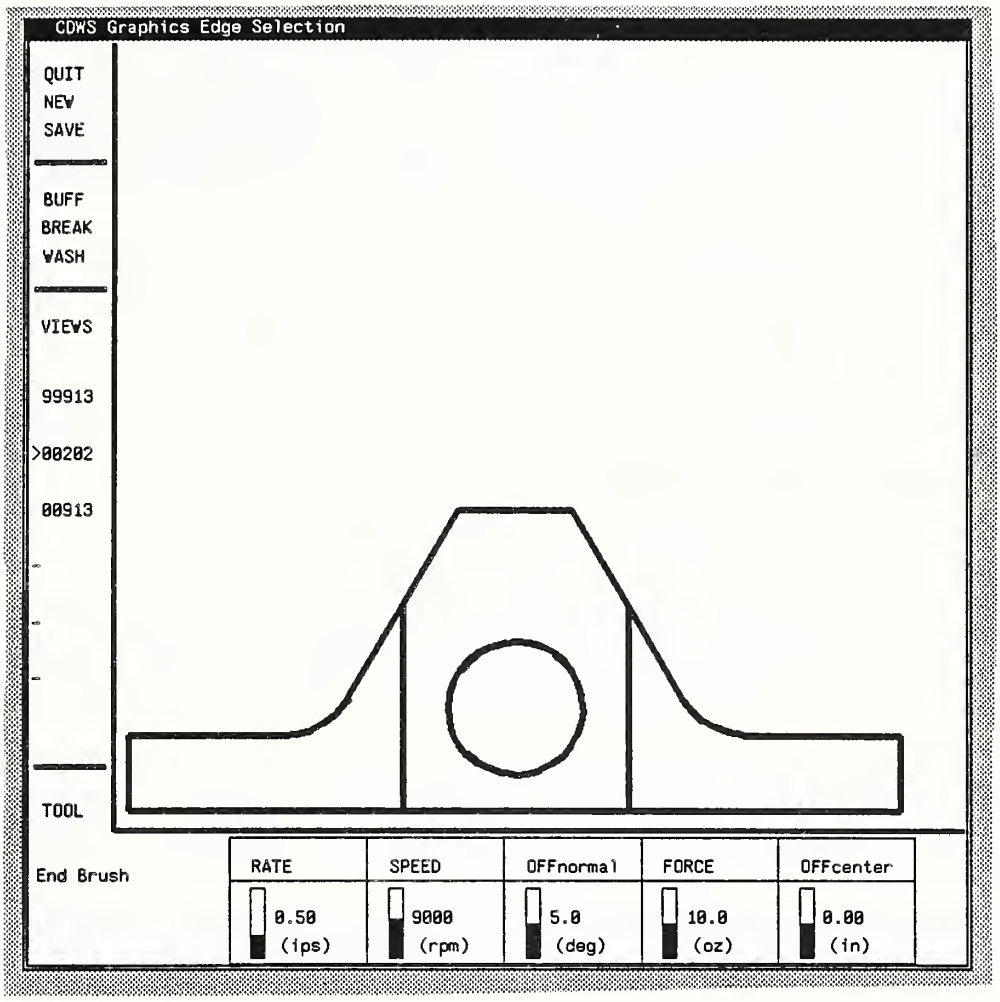


Figure 16. Deburring Selection Window

reorientations and to use the tools in the order of decreasing aggressiveness. When these rules are not desired, a BREAK forces the activities to be performed in the order inputted by the user. The WASH button causes a break and creates an association which directs the workpiece to the washer/dryer for cleaning.

The deburring planner is fairly primitive. For example, the user must avoid paths which cause non-working portions of the tool to strike the workpiece. Other shortcomings of the system are readily apparent during use. The program is written in the C language and the files are in the directory /usr/norcross/master/deburr.

2.2.8. Quit

The eighth menu element,"Quit", terminates the master program without confirmation. This element also signals the controller that the programming for this workpiece is complete. Thus if the master window is terminated prematurely, the controller should be suspended and the master command reissued. This may be accomplished by either issuing the pause command and calling master from a separate shell, or by issuing the command:

```
WSC> exec master [workpiece type]
```

This command also suspends the controller, but resumes the controller immediately upon completion of the master program.

2.2.9. Help

The on-line help function is not currently installed. In response to this command, the system directs the operator to an informed user.

3. ADDING PLANNING FUNCTIONS

One of the earliest goals of the Cleaning and Deburring Workstation was to implement online programming [16]. Planning may be used to enhance the flexibility of almost any application. Currently the workstation uses on-line planning for tray decomposition and for workpiece coordination. Other areas where planning would be useful are movement planning and error recovery.

The implementation of a planning scheme requires a planning function and an executing decomposition plan. The planning function should be an executable Unix shell command. The function should return a readable disk file which contains a decomposition plan in the "add_plan" format discussed earlier. The decomposition plan must: 1) call the planning function, 2) monitor the disk for the return, 3) load the plan into the executing lisp environment, and 4) execute the plan. The decomposition may also remove the plan after its work is completed. The planning function can be executed within the WSC's lisp environment but that may unacceptably slow the actions of unrelated tasks.

The application of planning routines is virtually unlimited. Explaining the possible alternatives is impractical. However the executing decomposition plan will have the basic format shown below.

```
(add_plan plan_it
          (()
           ( ... )
           ((1 (0) ((set-job-data 'plan_name (gensym)))
               INSTRUCTION exec ('plan_it ($$ plan_name) ...))
            (2 (1) ((probef ($$ plan_name))) () NOP ())
            (3 (2) ((load ($$ plan_name))) () NOP ())
            (4 (3) () MACRO execute (($$ plan_name)))
            (5 (4) () INSTRUCTION report ())))
```

Figure 17. Sample Executing Decomposition Plan

In the example in figure 17 there are no resources listed. If the planning function precludes multiple executions, plan_it would also have to be a flag-type resource to prevent multiple calls. The arguments to the planning function are application dependent but should be complete since access to the WSC's or the AMRF's data is not available.

Appendix A: CDWS-WSC OPERATIONS SUMMARY

LIGHT-OFF & INITIALIZATION

1) If SUN terminal is running "Game Of Life"; press any key and go to step #4.

2) Else, turn the SUN computer on via the three main power switches located on the disk pedestal (at right side of desk), the base pedestal (behind desk) and the back of the terminal screen.

3) Log into the system with: **`cdws login:`** `rjn`

4) Start the controller program by:
   open rootmenu
      position the mouse arrow over the gray area
      open by depressing right mouse button
   open "WSC" submenu
      with right button depressed, move mouse to the "=>" to the right of "WSC"
   highlight appropriate configuration with mouse arrow

w/ prompts	controller prompts for CMM & robot status
fully config	CMM at "demo", database & both robots running
CELL test	CMM at "demo", database & robots not running
dBase test	CMM at "demo", database running, robots not running
standalone	no CMM, both robots running
760 only	no CMM, Unimate 2000 not running
cmm	(ignore)

5) Wait. The system takes about three minutes to initiate and run.
   The prompt **WSC>** will appear in the terminal window.

TERMINAL INPUTS

To type into the terminal window the mouse arrow must be in that window.

Commands are entered by typing the command followed by its parameters. E.g.:

```
WSC>  teach block_fh 1 1 0
```

When the system asks a question it will provide an identifier which must be the first element of your response. Also, the system generally gives a guide to the acceptable responses. For example:

```
WSC>  [j5] Save this Process Plan? <n/number>
WSC>  j5 n
```

SHUTDOWN

1) Enter: **WSC>** `quit`
2) Wait. "Game of Life" will appear on the screen.
3) If desired, turn off the three power switches.

Appendix B: CDWS-WSC COMMON COMMANDS

RECEIVE_TRAY {[TRAY_SER_NR]} {[TRANSFER_POINT]}
locks the transfer point tray station and fetches the *Tray Contents Report* from the data base.
TRAY_SER_NR is the "tray serial number" and defaults to "mt_tray"
TRANSFER_POINT is the AMRF name for the tray station and defaults to "CDWS_TP1"

DEBURR_LOT [LOT_ID] {[OPERATION_SHEET]} {[LOT_TYPE]}
deburrs the lot of parts as specified by the instruction_set declared in the operation_sheet.
LOT_ID is the lot's identification number and is a required entry
OPERATION_SHEET defaults to the standard instruction_set numbers
LOT_TYPE is the part description and defaults to "MIXED" which means "do all parts"

SHIP_TRAY {[TRANSFER_POINT]}
releases the transfer point tray station and clears those parts from the internal data base

(Note: The previous three commands are the standard sequence from the external command source)

teach [part-type] {[sector]} {[tray]} {[location]}
develops and tests a process plan for the specified type of part.
part-type is the part description and is a required entry
sector is the tray sector where the part is buffered and defaults to "1"
tray is the tray station where the part is buffered and defaults to "1"
location is the current "vise" position of the practice part and defaults to "TRAY"

move-part [part] [goal]
moves the named part to the goal position. The part must be in the internal data base.
part is the name of the part and is required
goal is either "TRAY" or a vise-grip location where the part is to be placed and is required

add-part [part] [part-type] [tray-station] {[location]}
adds a part to the internal data base.
part is the name of the part and is required
part-type is the part description and is a required entry
tray-station is the buffer position of the part (e.g. TRAY11,TRAY24) and is required
location is the current position of the part and defaults to the buffer position

set-location [part] [location]
changes the part's location in the internal data base (w/o part movement)
part is the name of the part and is a required entry
location is the part's new position (must specify "TRAY11" vise the generic "TRAY")

RCS [command-string]
sends the command to the task level of the RCS controller controlling the PUMA 760.
command-string is the command to be sent and must be enclosed in double quotes (" ")

VAL [command]

sends the command to the VAL+ controller controlling the Unimate 2000.
command can be multiple words and doesn't require quotes (e.g. `VAL do depart 150`)

`re-initialize_VAL`
reestablishes the command/status handshake with the Unimate 2000. This command should be executed when the Unimate is placed in local mode and when the Unimate is returned to WSC's control.

`deburr-part [part] {[instruction_set]}`
deburrs the specified part based on the instruction_set.
part is the name of the part and is a required entry
instruction_set is the process plan to be used and defaults based on the part type

`deburr-loops [part-type] [loops]`
commands the PUMA 760 to deburr the specified loops. A workpiece of the given type must be properly positioned in the vise and the appropriate RSL file downloaded to the PUMA's RCS controller. E.g. `deburr-loops pipeclamp_fv (0 1 2)`
part-type is the workpiece description and is a required entry
loops is the list of loop numbers which coorelate to the RSL file

`RCS_status`
returns the most recent status from the RCS controller controlling the PUMA 760.

`initialize_dbase`
generates the UVA protocol for establishing communications with the AMRF data base.

`set db-flag nil`
shifts the controller to look at local copies of data rather then the AMRF data base.

`initialize_CELL`
establishes communications with the CMM and in turn with the remote command source.

`set CMM nil`
causes the controller to ignore the remote command source.

`initialize_RCS`
establishes communications with the RCS controlling the PUMA 760.

`initialize_VAL`
establishes communications with the VAL+ controller controlling the Unimate 2000.

`set VAL-flag nil`
breaks the communications with the VAL+ controller controlling the Unimate 2000.

Appendix C: CDWS-WSC OPERATING INSTRUCTIONS

1. SYSTEM INITIALIZATION

The Cleaning and Deburring Workstation has several methods of initialization. The most common method is outlined in Appendix A. The technique in that appendix instructs the system to automatically perform several of the start up procedures. This section describes the step by step methods.

1.1. Machine Booting

The CDWS workstation controller executes on a SUN3/160M computer workstation which is located in the south-east corner of the workstation. The SUN computer is designed for continuous operation and is normally left running between activities. When running unattended, the SUN executes a screen saver program known as "The Game of Life". This program flashes patterns of SUN logos on a darkened terminal screen. To clear the "Game of Life" depress any key. The computer is usually in an appropriate state to run and modify the workstation controller program

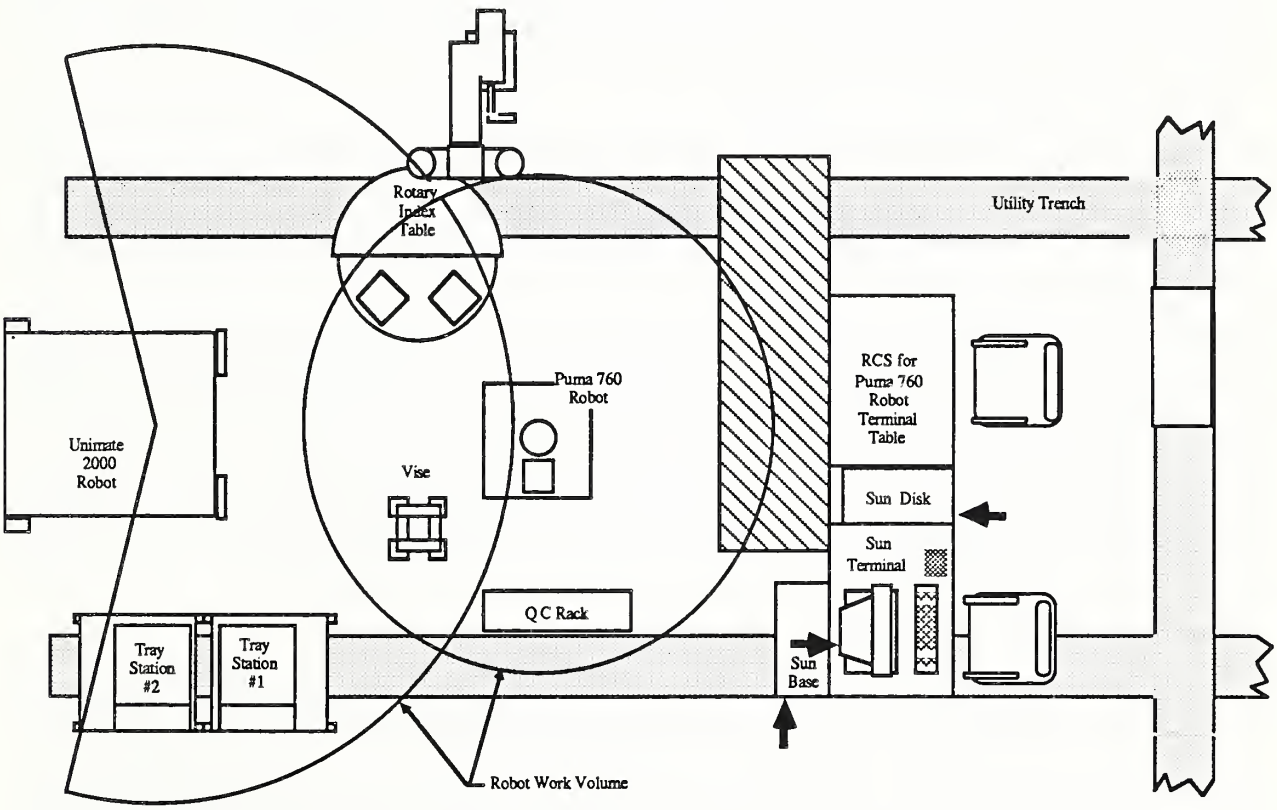


Figure C-1. CDWS Layout With SUN Switch Locations

and does not require booting. Section 1.3 should be attempted before rebooting the computer.

Occasionally the system will be running a user which cannot execute the workstation controller program. To clear the user enter "^D^Q", which clears the Suntool interface, and "^D", which logs out the old user. The SUN will then display a login prompt as described below. If this procedure fails to interrupt the current user the unit may be rebooted by toggling the on/off switch on the computer pedestal or by simultaneously depressing the "L1" and "A" keys.

To power up the SUN computer, turn the disk, the computer, and the screen on. The power switches are located on the respective units (dark arrows in Figure C-1) and are marked "1" for on and "0" for off. The SUN computer automatically boots the Unix operating system at power up. Booting Unix takes several minutes, produces between two and three pages of messages, and ends with the login prompt:

cdws login:

If booting stops before the login prompt refer to the subsection of this manual on errors.

1.2. Login

To enter the computer as a user able to execute the workstation controller, login as "**cdws login:** rjn" (there is no password required). The SUN automatically starts the suntool interface with one small terminal window and several icons in the lower right corner. The SUN is now ready to run the workstation controller program.

1.3. Starting the Program

The workstation controller program is in "/usr/local/norcross/wsc/wsc". Therefore the controller can be executed by opening a shelltool with directory "/usr/local/norcross/wsc" and entering "**cdws-#>** wsc". An alternative method is to expand the rootmenu "WSC =>" entry and select the appropriate configuration.

The wsc program takes four optional arguments which establish the communications desired when executing the controller. The arguments are; the Common Memory Manager (CMM) location, and predicates for connecting to the AMRF data base (IMDAS), the PUMA 760, and the Unimate 2000. The CMM generally runs on the SUN workstation named *demo* which is located in Control Room II. The Unix /etc/hosts file contains an entry "CMMremote" as an alias to "demo". The system administrator moves this alias to follow the current default CMM location. Thus "CMMremote" is generally the correct location. For the predicates the entry should be either "t" or "nil" to indicate whether the corresponding component is operational or not respectively. If an argument is omitted the controller will ask for the status when it is initializing.

The expanded rootmenu provides six alternative configurations. When using the rootmenu expansions the system generates its own windows and shells. The first rootmenu alternative, "**w/prompts**", is similar to executing "wsc" without any arguments, i.e. the controller prompts the user for the information. The second alternative, "**fully config**", puts the CMM at "CMMremote" and has the database and the PUMA 760 and Unimate 2000 robots running. The third alternative, "**CELL test**", has the CMM at "CMMremote" with no database and neither

robot running. "**dBase test**", the fourth alternative, sets the CMM again at "CMMremote" and establishes links with the AMRF database. The final two alternatives, "**standalone**" and "**760 only**", assume no CMM and either both or only the PUMA 760 robot running. The last entry in the expanded menu, "**cmm**", generates a CMM at the cdws workstation but does not execute the workstation controller program. Table C-1 lists the rootmenu entries and their command equivalents.

Menu Selection	**Command Equivalents**
w/ prompts	**cdws-#>** wsc
fully config	**cdws-#>** wsc CMM_remote t t t
CELL test	**cdws-#>** wsc CMM_remote nil nil nil
dBase test	**cdws-#>** wsc CMM_remote t nil nil
standalone	**cdws-#>** wsc nil nil t t
760 only	**cdws-#>** wsc nil nil t nil

Table C-1. "wsc" Expansion Menu Command Equivalents

1.4. Establishing Communications Links

Each communications link is established by a unique procedure. Normally the interface is created at start-up and can be ignored by setting flags. Separate procedures initiate the communications links while the controller is running.

The Common Memory Manager establishes and maintains mailboxes for the External Command Source and the AMRF data base (IMDAS). The workstation command "**WSC>** `initialize_CELL`" establishes communications to the CMM. In response, the controller freezes and requests the location of the CMM. The command "**WSC>** `set CMM nil`" breaks the logical connection to the CMM. The physical link to the CMM is an Ethernet. Failure of the ethernet (e.g., a disconnected wire) causes communication failure.

Although IMDAS communicates command and status through the CMM, there is a separate handshake established for the database. The command "**WSC>** `initialize_dbase`" establishes this handshake. Likewise the command "**WSC>** `set db_flag nil`" breaks the handshake. When the handshake breaks the workstation controller automatically shifts to local copies of data.

The command "**WSC>** `initialize_RCS`" sets and breaks the communications link with RCS controller. Like the CMM connection, this command freezes the controller while the system inquires about the "task" link and the "RSL" link.

The communications link to the VAL controller can be established by either of two commands; "**WSC>** `initialize_VAL`" or "**WSC>** `re-initialize_VAL`". The former freezes the controller for questions while the later does not. To break the link to the VAL controller enter

"**WSC>** `set VAL-flag nil`". Table C-2 lists the connect and disconnect commands for each of the communications links.

Communication Link	Connect	Disconnect
External Command Source	**WSC>** initialize_CMM	**WSC>** set CMM nil
AMRF Data Base (IMDAS)	**WSC>** initialize_dBase	**WSC>** set db_flag nil
RCS controller	**WSC>** initialize_RCS	**WSC>** initialize_RCS
VAL controller	**WSC>** re-initialize_VAL	**WSC>** set val-flag nil

Table C-2. Communications Connect and Disconnect Commands

2. ERRORS

There are several types of errors which may be encountered when operating the system. The most disastrous errors are those affecting the SUN workstation, the Unix operating system, and the Franz Lisp programming environment. These problems will always shut down the system and all data will be lost. When the execution of a particular command is interrupted the effects are not as far reaching but care is required to prevent the error from cascading into a more serious situation. Deadlock and livelock detection and recovery pose a particular problem. However these situations have not been present in testing and are lightly covered in this manual.

2.1. SUN Computer System Errors

The SUN workstation used in the Cleaning and Deburring Workstation is not designed for an industrial environment. Therefore at unpredictable intervals the SUN will cease to function. Current theory holds that dust accumulates on the processor board and the disk controller. When this problem occurs the SUN monitor will declare errors such as "dsk hrd error" or "bad blk not found". To correct the problem; turn off the SUN, remove the VME boards, clean the air filters and computer boards with air and a brush, clean all contacts with a contact cleaner, and reassemble the machine. If this does not work, call SUN maintenance for help (1-800-USA-4SUN).

The Unix operating system on the SUN is also not designed for this application. The Unix kernel maintains a record of much of the activity in the workstation. These records are maintained in the root partition of the disk. Eventually the root partition becomes full and the operating system reports a problem such as "file table full". After this command the state of the control system is unpredictable. To correct; turn the SUN workstation off and back on. This action clears the records which actually aren't required.

2.2. Guided Error Recovery

There are three types of errors; those which are predictable but do not confuse the world model, those which are predictable and do confuse the world model, and those which are unpredictable. The world model is the data held in the controller and used to schedule and plan the activities of the

system (e.g. workpiece location). The world model becomes confused when an error occurs during the transition of an element of that data (e.g., workpiece movement). An error is unpredictable when a solution for that error has not yet been determined. A robust control system requires the ability to handle each type of error.

Errors which do not confuse the world model are handled in the decomposition plan. Since the effects of these errors by definition do not effect other tasks, the recovery is implicitly coded into the decomposition plan. Thus these errors are more alternative states then errors. Usually the solution is to restart the routine which failed.

The more serious situation is an error which confuses the world model. Normally these errors indicate that the world model is incorrect. The solution to these errors requires information such as what went wrong and what was the system trying to accomplish when the error occurred. In practice, sensors answer the first question while the decomposition hierarchy answers the second. The solution is to replace the decomposition plan where the world model data is inaccurate with another decomposition task which corrects the error. If the replacement task is unable to correct the problem the "prune and replace" procedure is repeated further up the decomposition.

A typical second type error is when a robot transfers a workpiece and finds its path blocked. The supervisory controller backs up the decomposition from monitoring robot status to where the movement decision was made. Sensors indicate that the robot is holding the workpiece and the robot is waiting near the delivery point. To recover, the system attempts to clear the delivery point. If the next attempt to deliver the workpiece fails, the system backs up the decomposition to where the delivery point was selected and attempts to select a different delivery point. Once successful, the world model data is modified and the execution resumes. If the back tracking reaches the root of the decomposition tree then the error is considered unpredictable.

Unpredictable errors are ones which have no known recovery. Handling an unpredictable error consists of shutting down a minimal portion of the system and requesting operator intervention. A common unpredictable error is missing or incomplete data. Here the system expunges the task without any system reduction. The portion of the system which is shut down is defined by the resources held in the decomposition tree. System response to an unpredictable error is necessarily robust, which leads to laborious manual recoveries.

As a system matures, the solutions to many "unpredictable" errors are determined and implemented into the system. An important aspect is that each "predictable" recovery is another decomposition plan.

2.3 Deadlock Detection and Recovery

Since the workstation controller schedules tasks based on resource allocation there exists the possibility of creating a deadlock condition. A deadlock exists when two or more tasks are waiting for resources which are held not by their parent, but by the parent of one of the other tasks. The controller has no mechanism installed for detecting or correcting deadlock. In the current implementation there is only one shared resource, the VISE, therefore deadlocks are impossible (a minimum of two such resources are required). As the workstation develops other shared resources

will be added. At that time the workstation programmer must avoid deadlocks through the careful crafting of the decomposition plans.

Deadlocks are detected by a lack of progress on a set of tasks. Recovery consists of terminating one of those tasks and restarting it once the other task has completed. Again this situation should never occur.

3. SYSTEM INPUTS

The system recognizes two types of inputs from the terminal, commands and responses. Commands include task initialization commands and monitoring commands. Responses are answers to questions posed by the system to an operator. This section explains the format of these inputs.

Although the controller is written in Franz Lisp, the terminal interface does not require an operator to possess any knowledge of the Lisp programming environment. The system accepts commands in prefix format. The command string consists of the command name followed by parameters or parameter pairs. Parameter pairs are the parameter name and value enclosed in parentheses. When parameters are not provided in pairs they must be in the proper order. The system displays the proper parameter order and the default value in response to the syntax command. When the operator omits a parameter the system inserts the default value. The format for the syntax command is:

```
WSC> syntax [command name]
```

Thus since:

```
WSC> syntax DEBURR_LOT
(lot_id nil) (part_type MIXED)
```

the following four inputs are equivalent:

```
WSC> DEBURR_LOT 415
WSC> DEBURR_LOT 415 MIXED
WSC> DEBURR_LOT (lot_id 415)
WSC> DEBURR_LOT (part_type MIXED) (lot_id 415)
```

A "nil" default value at the beginning of the list usually indicates that the parameter is required for execution. A "nil" default value at the end of the parameter list indicates that the system will generate an appropriate value for the parameter based on the other values.

To monitor the execution of the system, the operator has direct access into the Lisp environment. To execute a Lisp command the Lisp format is used. For example:

```
WSC> (setq fact '(SIMPLY AMAZING))
WSC> (print fact)
(SIMPLY AMAZING)
WSC>
```

Extreme caution is advised when directly accessing Lisp. The system uses several global variables. Changing one of these may produce unpredictable behavior (although most likely the system will just stop).

During execution of some commands the system may request additional information or actions from

the operator. These requests appear in the terminal window and consist of three parts; an identifier, the request, and an answer guide. The identifier is the letter "j" followed by a number enclosed in square brackets (e.g. **[j31]**). The request is an Ascii string requesting data or stating instructions for the operator. The answer guide is optional and appears within angled brackets following the descriptive string. The answer guide lists alternative responses separated by slashes (e.g. **[y/n]** for yes or no).

To respond, enter the identifier followed by the answer. If no answer is required, simply enter the identifier. For example:

```
WSC> [j23] Save Process Plan? [n/number]
WSC> j23 n
```

and

```
WSC> [j51] Move Unimate 2000 to safe location.
WSC> j51
```

4. BASIC SYSTEM COMMANDS

This section explains some of the basic commands currently running in the system. The commands are separated by function. Many of the commands rely on data generated by other commands and understanding these interactions is necessary to properly operate the workstation.

4.1. Part Movement

The workstation controller is designed to permit, and encourage, real-time planning and opportunistic scheduling. Thus the system relies on a substantial amount of data to plan the movement of parts through the workstation. If this data is incorrect, the resulting plan will probably cause an error, specifically a collision between parts.

The command to move a part is "move-part". Move-part takes two arguments; the part's name and the destination point for the part (see appendix B). Other significant data, such as the part type and the part's current position, is taken from a local data base. The contents of this data are shown in response to the "parts" command as per figure C-2.

```
WSC> parts
The parts in the workstation are:
P_CLAMP_1   pipeclamp_fv   (CDWS_TP1 . 1)   (VISE . 5562341)
P_CLAMP_2   pipeclamp_fv   (CDWS_TP1 . 2)   (CDWS_TP1 . 2)
P_CLAMP_3   pipeclamp_fv   (CDWS_TP1 . 3)   (CDWS_TP1 . 3)
P_CLAMP_4   pipeclamp_fv   (CDWS_TP1 . 4)   (CDWS_TP1 . 4)
Trays in the workstation are:
((tray_420 1))
WSC>
```

Figure C-2. Example Workpiece Location Display

The "parts" command displays the current listing of parts in the workstation. For each part the controller displays the name, type, home tray location, and current location of the part. The

command also displays the trays currently in the workstation. The trays are shown as element pairs of name and location (tray station one or two).

If a part is not reflected in the data display, it should be added with the "add-part" command. The "add-part" command takes three required and one optional arguments. The required arguments are the part's name, type, and home tray location. The optional argument is the part's current location which defaults to the home tray location. The tray location is expressed in a concatenation of "TRAY", the tray station, and the tray sector number (e.g. TRAY13, TRAY25, ...). The current location is expressed as a tray location or a vise location. The vise locations are numerical references to the entries in the fixture files for the part (as found in "/usr/local/CDWS/parts/[part]/vise*" files).

If the "parts" data display shows the part in an incorrect location, the data can be updated with the "set-location" command. The "set-location" command requires two arguments; the part name and the part's true location. The location is represented as described above.

In normal operation the "parts" data is maintained automatically through the "RECEIVE_TRAY", "SHIP_TRAY", and "move-part" commands. Thus corrections are only required after unpredicted errors.

Once the data is correct, the part can be ordered about the workstation with the "move-part" command. Based on the example workpiece locations given in figure C-2, the command

```
WSC> move-part 5562973 P_CLAMP_1
```

would cause the part "P_CLAMP_1" to be reoriented. While

```
WSC> move-part 5562341 P_CLAMP_2
```

would wait until the vise became free. Since the controller's world model contains the workpiece's default tray position, the goal argument can be just "TRAY" and the system would place the workpiece in the appropriate position.

4.2. Transfer Point Interface

The Transfer Points, or tray stations, are shared resources between the Cleaning and Deburring Workstation and the Material Handling Workstation. Since the workstations may not use the tray stations simultaneously, there is a protocol for locking in one workstation and locking out the other. The actual interface to the tray stations is not addressable from the workstation controller. Instead commands are routed through the VAL II controller.The tray transfer commands, "RECEIVE_TRAY" and "SHIP_TRAY", have components which command tray lock-in and lock-out.

The low level command is a VAL I/O Board command. Table C-3 gives the commands to set the access control to the tray stations. There is no positive response from the VAL+ controller to indicate success or failure. However the request is repeated automatically until the lockout occurs and the Unimate 2000 does not approach the tray station until it occurs. This interface is a potential source of problems but has been relatively successful so far.

DESIRED ACTION	COMMAND
Control Tray Station #1	WSC> VAL SIG l.u.req.ts1
Release Tray Station #1	WSC> VAL SIG l.u.rel.ts1
Control Tray Station #2	WSC> VAL SIG l.u.req.ts2
Release Tray Station #2	WSC> VAL SIG l.u.rel.ts2

Table C-3. Tray Station Control Commands

The high level tray transfer commands, "RECEIVE_TRAY" and "SHIP_TRAY", perform tasks other then tray station control. The "RECEIVE_TRAY" command queries the remote data base and updates the internal data base as to the parts in the workstation (see previous section). The "SHIP_TRAY" command clears this data.

4.3. Deburring Commands

The primary function of the Cleaning and Deburring Workstation is deburring parts produced in the AMRF. Therefore there is a set of commands that perform the deburring function. These commands generally call each other and are listed here in their hierarchal order.

"DEBURR_LOT" is the highest level deburring command. This command has one required argument; "LOT_ID". (See Appendix D for the current part-type/lot-id/tray number correlations.) "DEBURR_LOT" fetches the lot description from the remote data base and deburrs the individual parts of that lot.

```
WSC>  DEBURR_LOT 420
```

Each part in a lot is deburred via the "deburr-part" command. This command requires a part name to execute and is the core of the deburring function of the workstation. The command fetches data, converts geometry files, generates robot trajectories, maintains the RSL memory, and directs the teaching and reteaching of the deburring paths. The command also directs the generation of robot coordination plans and uses those plans to control the workstation.

```
WSC>  deburr-part P_CLAMP_1
```

When a part is in the vise, the command to deburr the edges on the exposed surface is "deburr-loops". This command requires a part-type and a list of loops to be deburred. The loops correlate to the RSL trajectory file. This command also generates reteach commands as required.

```
WSC>  deburr-loops pipeclamp_fv (0 1)
```

The lowest level deburring instruction is the command to the RCS controller. This command takes the same arguments as the "deburr-loop" command but the form is different.

```
WSC> RCS "DEBURR_LOOP pipeclamp_fv 0 1;;"
```

4.4 Data Base

The controllers at the AMRF are designed to be data driven. This data resides in local disk files and in a remote data base. Access to the data is via processes that are defined in decomposition plans.

Normally the system looks to the remote data base for the data. However if that data base has trouble with a query or takes more then ten minutes on a query, the system automatically shifts to the local copy of the data. The part names in the local data are different from the part names in the remote data. Thus if the shift occurs in the middle of a task it may cause an error.

Communication is established with the remote data base through the UVA-DataBase protocol. For more information on communications with the remote data base, please see the section on Communication interfaces.

4.5 Basic Demonstrations

The AMRF commonly runs demonstrations for various public groups. Very few of these groups understand or are interested in the inner workings of the various control structures in place at the AMRF. Therefore various stand alone demonstrations are "canned" to give these groups a flavor for the capabilities of and the work going on at the AMRF. The Cleaning and Deburring Workstation has two of these demonstrations.

The more general demonstration simulates the command sequence generated at the External Command Source (ECS); "RECEIVE_TRAY", DEBURR_LOT", "SHIP_TRAY". The command "demo" generates this sequence with the appropriate argument values. The "demo" command takes two arguments; part-type and tray-station. The part-type can be "pipeclamp_fv", "block_fh", "dog", "shuttle", or "lclevis". The tray-station is either "1", for the tray closer to the controller, or "2" for the more distant tray. This demonstration teaches a part-type once and deburrs the remainder. The workstation reteaches a part-type after a given number of parts. This interval is set to 20 by default but may be changed by:

```
WSC> set teach-interval 10
```

or any other integer number. Example:

```
WSC> demo block_fh 1
```

A more interesting demonstration is the local generation of the part's Process Plan. Unlike much of the AMRF, the Cleaning and Deburring Workstation does not utilize a set of hierarchal process plans to perform its functions. Instead there is a single process plan for the part and the various levels (e.g., workstation, PUMA 760 robot, Unimate 2000 robot, ...) plan their various processes on-line. The format of this single process plan follows the published AMRF Process Plan Format but is generated through a graphics interface at the workstation.

The demonstration of this procedure is its execution. The "teach" command executes the plan generator and tests the plan on a part in the workstation. The command takes one required and three optional arguments. The required argument is the part-type. The optional arguments specify the current location for the part of that type and where to leave the part when the planning is complete. The arguments are the sector number, the tray station number, and the current location. The command defaults these parameters to sector "1" of tray station "1" with the part at "TRAY11". The location argument always defaults to the home tray location. The command generates the appropriate "RECEIVE_TRAY" and "SHIP_TRAY" commands as required. Example:

```
WSC>  teach dog
```

For more on this procedure refer to the portion of this manual dealing with adding new workpieces.

Completion of the plan generation automatically begins a test of that program. The testing proceeds similar to the "deburr-part" command mentioned above. However the edges are always taught. The system asks if it should retain the new process plan, for demonstrations the answer should be no.

```
WSC> [j3] Save new process plan? [n/number]
WSC> j3 n
```

4.6 General

All of the system's commands are not listed in this section. A fuller list can be found in Appendix B of this manual. All decomposition commands in the system can be found by entering:

```
WSC> plans t
```

into the main terminal window. However the system is also designed to accept all valid Lisp commands and to display global variable values as if they were commands, therefore it is impossible to list all the possible commands. The commands listed here and in Appendix B should be sufficient for system usage and limited expansion.

5. SHUTDOWN PROCEDURE

The workstation controller has several shut-down commands. The effect of these commands ranges from temporary interruption to initiating the Sun's screen saver program. None of the shut-down commands issue special instructions to the components of the workstation. Thus the workstation's robots will continue to execute their last command.

The "pause" command generates a temporary interruption in the workstation controller. The command jumps the controller program into a Lisp "read/eval" loop. Any of the system's global variables may be modified while in the "pause" loop. This command is commonly used to stop decomposition while the local data is verified and modified. Figure C-3 shows a "pause" session.

```
WSC> pause
Workstation Controller interrupted
Enter "RESUME" to continue.
pause> (setq a 1)
1
pause> a
1
pause> RESUME
WSC>
```

Figure C-3. Sample Pause Sequence

The workstation controller program is halted with the Lisp environment intact in response to the "halt" command. All of the controllers sub-processes are terminated before the Lisp environment is returned to the operator (takes about a minute). This includes the diagnostics windows and the communications links with the workstation robots and the common memory manager (CMM). The operator cannot restart the controller program after issuing a "halt" command but all of the

controller's variables are available for inspection.

```
WSC> halt
halted on request
=>
```

An "exit" command stops the controller like the "halt" command and terminates the Lisp environment. This command leaves the operator in suntools for further work on the Sun computer.

```
WSC> exit
cdws-#>
```
{the terminal screen may disappear}

The "quit" command terminates the environment like the "exit" command but it also starts the screen saver program, "Game of Life". Since the Sun computer is designed for continual operation this is the normal method for session termination.

```
WSC> quit
```
{the Game of Life starts}

Appendix D:
DEFAULT PART TYPE/LOT ID/TRAY CORRELATIONS

The AMRF IMDAS maintains a set of standard data which is used for testing. The CDWS maintains a portion of this data on the disk of the SUN computer. Table D-1 lists the part-type to processing data correlations. When operating the workstation, the user should refer to this table for parameters to the RECEIVE_TRAY and DEBURR_LOT commands.

part-type	LOT_ID	TRAY_SER_NR
dog	400	tray_400
dog	405	tray_405
block_fh	410	tray_410
block_fh	415	tray_415
pipeclamp_fv	430	tray_420
pipeclamp_fv	435	tray_425
lclevis	440	tray_440
lclevis	445	tray_445

Table D-1. Test Data Correlation

The test data is located in directory /usr/local/CDWS/data/db_local. The tray contents report files are named the same as the tray serial number. The lot status report file names are formed by concatenating "lot_" with the lot id.

Appendix E: CDWS WSC-RCS INTERFACE

This appendix describes the interface between the SUN and the PUMA 760 at the Cleaning and Deburring Workstation. The section is divided into three parts: the interface to the RSL board; the interface to the Task board; and the communication protocol used by both boards.

1. RSL COMMAND INTERFACE

All lines sent by the SUN to the RSL board are executed as if they were typed at the terminal or loaded from a block. RSL or any recognizable commands can be sent. For example `3 D>M` can be sent causing RSL to reset itself.

All values are sent in ASCII. String variables can be up to 15 characters long and can be comprised of any ASCII characters except CR and SPACE; however, only printable characters are recommended. Integers are 16 bits. Floating point numbers have two formats:

E format:

```
SPACE -123.45E-67 SPACE
```

Free format:

```
SPACE -123.456 SPACE
```

The decimal point must be shown.

Send the desired deburring paths to the RSL board. To redefine a loop with new values send a path with the same part name and loop number to RSL. To delete an individual loop or all the loops on a part send one of the following to RSL:

```
DELETE-LOOP    ["part-name"]   [loop#]

DELETE-PART    ["part-name"]
```

While Task level is executing, RSL can define new loops, redefine old loops, or remove old loops as long as the loops are not referenced by the commands sent to Task.

2. TASK COMMAND INTERFACE

There are two new commands to Task: DEBURR and TEACH. To command the robot to deburr one or more loops send the command:

```
DEBURR ["part-name"] [loop#] {[loop#]} ... {[loop#]} ;;
```

To run the self-correction routine, send the command

```
TEACH [ part name ] [loop#] {[loop#]} ... {[loop#]} ;;
```

to the Task board. Currently only the end brush is taught. If loops with different tools are sent down, then they are run as normal deburr loops.

All lines sent to the Task board are sent directly to the Task input command buffer. Thus any legal Task command can be sent. Such as:

```
RESTART
PAUSE
QC
JOYSTICK
GOTO [obj] [grip#] [loc type] [loc name]
```

See RSL documentation in reference [*karl*] for specific parameters.

3. COMMUNICATION PROTOCOL

There are three RS-232 lines between the SUN workstation and the 760 RCS bucket. Line 1 is connected to the RSL board and lines 2 and 3 are connected to the Task Board. On line 1 the SUN sends -paths- and commands such as DELETE-LOOP to RSL. On line 2 the SUN sends commands to Task. On line 3 the SUN receives status from Task.

The protocol on the command lines 1 and 2 is:

The SUN sends STX one or more times.
When RCS echoes this, the SUN sends a complete line. A complete line can be the entire -path-, the -path- header, a complete -ppt-, or a command such as REMOVE-LOOP. The maximum line length is 1000 bytes.
RCS echoes each character.
The SUN checks the characters to see that they are the same.
If they are not correct, the SUN sends STX, waits for echo, and retransmits the line.
Once the line has been sent correctly, the SUN sends SI ETX.
RCS echoes these and begins processing the line.
When RCS is done and if there are no errors, RCS sends two LF and then one about every second until the SUN sends STX.
If there was an error and RCS is still alive, RCS sends "0".

In HEX, STX = 02, SI = 0F, ETX = 3, LF = 10, and "0" = 30.

The protocol on the status line 3 is:

Task sends a status byte each time the status changes and then sends the byte again about every second. The status bytes are:

01 if Task is executing and has control of the vise,
02 if Task is executing and does not have control of the vise,
04 if Task is done and has control of the vise,
07 if Task is done and does not have control of the vise,
[#] if Task has an error.

Where [#] is the error number plus 30 HEX. For example, error 7 would be sent as 37 HEX.

Hardware: 25 pin D-Plug, SUN Tx on pin 2, SUN Rx on pin 3, Gnd on pin 7.

LIST OF REFERENCES

[1] J.A. Simpson, R.J. Hocken, and J.S. Albus, "The Automated Manufacturing Research Facility of the National Bureau of Standards", Journal of Manufacturing Systems, Vol. 1(1) pp17-32, 1982.

[2] Furlani et al; "The Integrated Manufacturing Data Administration System (IMDAS)", to be published as NIST-IR, 1988.

[3] B. Thomas, CELL Controller Operations Manual, NBSIR 88-3789, May 1988.

[4] M. Potts and C McLean, "Control-Database Interface: Workstation Level Reports", Internal AMRF Memo, 9 October 1986.

[5] Franz Lisp Reference Manual Opus 43, Franz, Inc., Alemeda, CA., March 1987.

[6] K.N. Murphy, F.M. Proctor, and R.J. Norcross, "CAD Directed Robotic Deburring", to be presented at Second International Symposium on Robotics and Manufacturing Research, Education, and Applications, Albuquerque, 1988.

[7] P.F. Brown and S.R. Ray, NBS-AMRF Process Planning System, NBSIR 88-3828, 1988.

[8] B.R. Fox and K.G. Kempf, "Complexity, Uncertainty, and Opportunistic Scheduling", Proc IEEE Conference AI Applications, (Miami Beach, FL), 1985.

[9] R.J. Norcross, "A Control Structure for Multi-Tasking Workstations", Proc of 1988 IEEE International Conference on Robotics and Automation, pp 1133-1135, Philadelphia, PA, 1988.

[10] J.S. Tu and T.H. Hopp, Part Geometry Data in the AMRF, NBSIR 87-3551, April 1987.

[11] K.N. Murphy, Real-Time Control System Modifications for a Deburring Robot - User Reference Manual, NBSIR 88-3822, Aug 1988.

[12] Programming Manual: User's Guide to VAL II Version 2.0 (398AG1), Unimation, Inc., Danbury, CN., December 1986.

[13] S. Rybczynski, et al; AMRF Network Communications, NBSIR 88-3816, June 1988.

[14] D.R. O'Halloran and P.F. Reynolds, "A Model for AMRF Initialization, Restart, Reconfiguration, and Shutdown", NBS/GCR 88-546, May 1986.

[15] SUN Commands Reference Manual, Rev G, SUN Microsystems, Inc., Mountain View, CA., February 1986.

[16] H.G. McCain, R.D. Kilmer, and K.N. Murphy, "Development of a Cleaning and Deburring Workstation for the AMRF", Proc of Deburring and Surface Conditioning '85, 2:26-43, Oct. 1985.

READER COMMENT FORM

The Workstation Controller of the Cleaning and Deburring Workstation User Reference Manual

You may use this form to comment on the technical content or organization of this document or to contribute suggested editorial changes.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

If you wish a reply, give your name, company, and complete mailing address:

_____

_____

_____

_____

_____

What is your occupation? _____

Note: This form may not be used to order additional copies of this document or other documents in the series. Copies of AMRF documents are available from NTIS.

Please mail your comments to:

AMRF Program Manager
National Institute of Standards and Technology
Building 220, Room B-111
Gaithersburg, MD 20899

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** (See instructions)	1. PUBLICATION OR REPORT NO. NISTIR 89-4046	2. Performing Organ. Report No.	3. Publication Date FEBRUARY 1989

4. TITLE AND SUBTITLE

The Workstation Controller of the Cleaning and Deburring Workstation.

5. AUTHOR(S)

Richard J. Norcross

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The Cleaning and Deburring Workstation at NIST's Automated Manufacturing Research Facility employs two robots and numerous supporting equipment to wash, buff, and deburr discreet metal workpieces. This manual describes the workstation controller for perspective users and for researchers interested in expanding the workstation's capabilities. The manual specifies the general control problem and provides the theoritical foundation of the solution along with specific implementation details. These details include important data structures, programming formats, interfaces, and the current operation of the controller and workstation.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

AMRF; Cleaning and Deburring Workstation; Workstation Control; Discreet Event Control System; Automatic Planning; Automated Error Recovery;

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

69

15. Price

$14.95


AMERICAN BINDERY
2
5-94