

**AN OBJECT-ORIENTED SOFTWARE ARCHITECTURE
FOR
THE CONTROL OF FLEXIBLE PARTS FEEDING SYSTEMS**

**by
GREGORY CARYLEE CAUSEY**

**Submitted in partial fulfillment of the requirements
for the degree of Master of Science**

Thesis Advisor: Dr. Andy Podgurski

**Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY**

January, 2002

**Copyright © (2002) by
Gregory Carylee Causey**

An Object-Oriented Software Architecture
For
The Control Of Flexible Parts Feeding Systems

Abstract

by

GREGORY CARYLEE CAUSEY

As vision-based flexible parts feeders continue to make in-roads into the manufacturing arena, a unified methodology and architecture for their programming becomes increasingly important. This thesis examines an object-oriented software architecture for programming flexible parts feeding systems. The architecture is vertically segregated into levels of responsibility. High-level software objects oversee system start-up, general system operation, and shutdown. Mid-level objects are used to perform functions associated with sub-systems of the feeder: part presentation, part location, part retrieval, and parameter auto-adjustment. Server-level objects are responsible for encapsulating specific hardware. Complementing this vertical segregation, as one descends the hierarchy, is a progressively tightening alignment of software objects with physical hardware.

A new high-speed parts feeder has been constructed and serves as the testbed for the development of the software. It utilizes a series of conveyors for part presentation, a PC-based vision system for part location, and an Adept robot for part retrieval.

ACKNOWLEDGMENTS

A few people have been instrumental in enabling me to finish this thesis.

Andy Podgurski, my advisor in this work. Thanks for signing the paper work and otherwise standing back and letting me “run the show” on my own.

The rest of my committee, Mike Branicky and Gultekin Ozsoyoglu.

Mike Branicky provided lots of input and discussion in the auto-adjustment component of the software. Also, thanks for giving the rough draft a good read through. Your edits were appreciated and have made the final version much better than it would have been otherwise.

CAMP, Inc. If it wasn't for Bill Stellhorn at CAMP having the foresight to fund the project, it would never have gotten off the ground.

Roger Quinn, my past advisor from the mechanical engineering department. Thanks for arranging the post-doc support while I worked on this project. I could not have finished the project without the financial support.

Nick Barendt. Your time invested in proof reading this document is greatly appreciated. Also, your initial work on the vision server portion of the code is still evident. Getting that component of the system off the ground helped immensely.

All my friends and family who still wonder if I'll ever leave school. I guess I'm finished for now. (Yes Dana, the “victory lap” is complete!)

TABLE OF CONTENTS

LIST OF FIGURES	ix
LIST OF TABLES	xi
1 Introduction	1
1.1 Architecture for Flexible Feeders	3
1.1.1 Feeder Sub-Systems	3
1.1.1.1 Parts Presentation	3
1.1.1.2 Part Pose Determination	4
1.1.1.3 Part Retrieval	5
1.1.2 Philosophy	6
1.1.2.1 Vertical Hierarchy	6
1.1.2.2 Object Alignment	8
1.2 Previous Work	8
1.2.1 Physical Feeder Designs	8
1.2.1.1 Adept	9
1.2.1.2 Seiko	11
1.2.1.3 Hoppmann Robotics	12
1.2.1.4 Intelligent Automation Systems	12
1.2.1.5 Robotic Production Methods Inc.	13
1.2.1.6 Original CWRU Feeder	14
1.2.1.7 New CWRU Feeder	15
1.2.2 Feeder Control Systems	17
1.2.2.1 Adept	17
1.2.2.2 Intelligent Automation Systems	17
1.2.2.3 Original CWRU Feeder	18
1.2.2.4 Others	18
1.2.3 Feeder Control Software	19
1.2.3.1 Adept	19
1.2.3.2 Intelligent Automation Systems	21
1.2.3.3 Cimatrix	22
1.2.3.4 Original CWRU Feeder	22
1.2.4 Wrap-Up	25

1.2.5 Other Documentation Sources	25
1.3 Thesis Organization / Outline	26
2 Feeder Control Hardware	29
2.1 Overall Control	29
2.2 Vendor Supplied Hardware Controllers	30
2.2.1 Adept AWC Controller	30
2.2.2 Galil 1822	31
2.2.3 Matrox Meteor II	32
2.3 Time-Critical Connections	33
2.3.1 Horizontal Conveyor Encoder	33
2.3.2 Camera Trigger / Conveyor Latch	34
2.4 Control Hardware Layout Diagram	35
3 Overall System Architecture	37
3.1 High-Level	38
3.2 Mid-Level	38
3.3 Server-Level	39
3.4 Hardware-Level	40
3.4.1 Galil 1822	41
3.4.2 Matrox Meteor II	41
3.4.3 Adept AWC	41
4 High-Level Components	43
4.1 Remote User Interface (GUI)	43
4.1.1 Front-End	43
4.1.2 Code	46
4.1.3 Login/Logout Sequence	47
4.2 remote_user Object	48
4.2.1 Login	50
4.2.2 User Logout	50
4.3 main_control Object	53
4.4 System Start-up / Shutdown	56
5 Mid-Level Components	59
5.1 Asynchronous Threads	59
5.2 Part-Specific Objects	60
5.2.1 Constructors / Destructors	62

5.2.2	Common Functionality	63
5.2.3	New Part Introduction	63
5.3	Utility Objects	64
5.3.1	Conveyor Encoder Synchronization	64
5.3.2	Parts Queue	66
6	Sever-Level Components	69
6.1	Robot Motion Server	69
6.2	Robot Position Server	71
6.3	Vision System Server	71
6.4	Conveyor Motion Server	73
6.5	Digital I/O Server	75
6.5.1	Public Interface	76
6.5.2	Implementation of the Callback Functionality	77
7	Hardware-Level Components	83
7.1	Galil 1822 Motion Controller	84
7.2	Matrox Meteor II Frame Grabber	85
7.3	Adept AWC Controller	85
7.3.1	V+ Sockets	86
7.3.2	Overall Software Setup	87
7.3.3	Robot Motion Server Program	87
7.3.3.1	decode_and_do	88
7.3.4	Robot Position Server Program	90
7.3.5	I/O Server Program	90
7.3.5.1	io_notify	90
8	Results, Future Work, Conclusions	93
8.1	Results	93
8.1.1	Feeding Scenarios	93
8.1.2	Throughput	94
8.1.3	Auto-Adjuster PSO	94
8.2	Future Work	97
8.2.1	Improving Functionality	97
8.2.1.1	Error Handling	97
8.2.1.2	Communications	97
8.2.1.3	Abstract Server-Level Base Classes	98

8.2.1.4 Data Logging	98
8.2.1.5 Multi-Part Queue.	98
8.2.1.6 MIL Encapsulation	100
8.2.1.7 Utility Programs	100
8.2.1.8 Remote User GUI	100
8.2.1.9 Variable / File Naming Convention	101
8.2.2 Extending Capabilities	101
8.2.3 Intelligent Auto-Adjustment.	102
8.3 Conclusions.	103
9 Bibliography	105

LIST OF FIGURES

Figure 1-1: Generalized Control Architecture	7
Figure 1-2: The Adept FlexFeeder 250	9
Figure 1-3: Adept iFeeder Sub-System Breakdown	10
Figure 1-4: Seiko Feeder; Top Schematic and Views	11
Figure 1-5: Hoppmann Feeder FC-2500: Schematic and View (with Adept Robot)	12
Figure 1-6: Intelligent Automation Systems: FPF-2000	13
Figure 1-7: Original CWRU Feeder	14
Figure 1-8: Side Schematic CWRU Original Feeder	15
Figure 1-9: New CWRU Feeder: CAD View	16
Figure 1-10: New CWRU Feeder.	17
Figure 1-11: FPF-2000 System Controller Physical Layout	18
Figure 1-12: Original CWRU Feeder Controller Layout	19
Figure 1-13: Example Q&A Screen of the “Part Tuning Advisor”.	20
Figure 1-14: Control Panel, Status Panel, and Feeder Record Panel	21
Figure 1-15: Original CWRU Feeder: System State Diagram.	23
Figure 1-16: Original CWRU Feeder Control Scheme	24
Figure 2-1: Layout of the Feeder’s Control Hardware	35
Figure 3-1: Software Architecture Overview	37
Figure 4-1: Feeder GUI	43
Figure 4-2: User Has Control (left) and Someone Else Has Control (right)	44
Figure 4-3: System Status Area	45
Figure 4-4: User Input Area and Auto Mode Indicator	46
Figure 4-5: Message Window	46
Figure 4-6: User Login.	51
Figure 4-7: User Logout.	52
Figure 4-8: main_control Start-up and Shutdown Interaction Diagram.	55
Figure 4-9: State Transition: STOPPED → RUNNING.	55
Figure 4-10: main() Function Pseudo-Code	57
Figure 5-1: Asynchronous Threads Pseudo-Code	60
Figure 5-2: Part Specific Object Class Diagram.	62
Figure 6-1: io_interest Registered Thread Data Structure	78

Figure 6-2: I/O Server Start-Up Interaction Diagram	79
Figure 6-3: Input Notification Interaction Diagram.	81
Figure 7-1: V+ Side Robot Motion Server Pseudo-code	88
Figure 7-2: decode_and_do Pseudo-Code	89
Figure 8-1: Test Parts: Lenses and Squares	93
Figure 8-2: Auto Adjusting Algorithm Pull Behavior.	96
Figure 8-3: Auto-Adjusting Algorithm	96
Figure 8-4: Multi-Part Queue	99
Figure 8-5: Retrieval Time vs. Part Location.	103

LIST OF TABLES

Table 4-1: remote_user Public Interface	49
Table 4-2: main_control Public Interface	54
Table 5-1: Conveyor Encoder Public Interface	65
Table 5-2: Part Queue Public Interface	66
Table 6-1: Robot Motion Server Public Interface	70
Table 6-2: Robot Position Server Public Interface	71
Table 6-3: Vision System Server Public Interface	73
Table 6-4: Conveyor Motion Server Public Interface	75
Table 6-5: I/O Server Public Interface	77

1 Introduction

There has been a large shift in the paradigm of automated manufacturing in the past ten to twenty years. Previously, it was equated with dedicated systems producing large volumes of a single product at high speed. The overriding theory was that if a large enough quantity of products could be produced at a high enough rate for a long enough time, then the cost of the equipment could be justified and a significant portion of the market could be satisfied. Secondary benefits, including reduced labor and consistent quality, were added bonuses but not driving factors.

This view no longer holds true. Smaller lot sizes, shorter times to market, increased product quality, higher product mix, and lower manufacturing costs are typical of the requirements of a modern manufacturing facility; one in which traditional automation systems are not feasible. They simply take too much time to design, setup, and install, and carry too high a cost to be economically justifiable in light of new goals and requirements. Goals and requirements which now point toward a rapidly reconfigurable automation system capable of manufacturing a wide variety of products.

The repeated use of a basic infrastructure coupled with quickly exchangeable modules specific to each particular assembly has the potential to fulfill these new requirements. A basic infrastructure allows a capital investment to be recovered over the life of many different products while modules encapsulate hardware, software, and processes specific to a particular assembly. Together they create a flexible assembly system which may be rapidly reconfigured for the manufacture of vastly different products. Realizing such a goal requires all components of the assembly system (both hardware and software) to interact with each other through well-defined interfaces.

A crucial component of any assembly system is parts feeders. Unfortunately, parts feeders are also one of the most specialized components of such systems. While a modular approach could be employed which would allow specialized feeders to be quickly brought into the workcell, there are several difficulties which present themselves using this approach. First, a feeding system tends to be large and bulky in comparison to other specialized components, such as a robot's gripper. Size alone could make storing all the specialized feeders difficult. Second, a feeding system is generally more expensive than

other specialized components. It might be difficult to economically justify building several new feeders for each new product being assembled. Third, the lead time to build and adjust most current feeding systems is rather long. This diminishes the ability of the workcell to be rapidly adapted to new products. Therefore, simply placing current feeding technology behind a generic, modular interface is not a feasible solution.

While most components of automated manufacturing systems have evolved to meet the increasing manufacturing challenges, parts feeding technology has been an area in which little progress has been realized. Vision-based flexible parts feeding systems are promising examples of emerging technology targeted at this field. In contrast to traditional feeding methods, such as bowl feeders, which use mechanical elements to singulate parts, flexible feeders use general sensing techniques (usually machine vision) to locate parts in conjunction with a programmable manipulator (usually a robot) for part feeding. This enables them to feed a wide variety of parts by simply changing the algorithm used to locate parts and the gripper used by the robot to handle the parts.

However, with this added flexibility comes additional complexity in operating the feeder. Traditional feeders generally have a single switch; turn them on and parts appear at a known output location and orientation or *pose* (location plus orientation equals pose). A single sensor could be used to determine when a part was present for retrieval. A flexible feeder is much more complex. Each feeder is like a mini-factory; parts are quasi-singulated[§] and presented in random poses, a sensor is used to determine which parts are in the proper pose for retrieval and subsequent assembly, and a programmable manipulator is instructed to retrieve the identified parts. Often, these sub-systems are controlled by separate processors, which must communicate with each other to enable the feeder to work.

It becomes apparent that in order to reach the desired goals of a modern assembly facility, new feeding technologies need to be developed. Flexible parts feeders can fulfill this need, however, such feeding systems are much more complex to program

[§]Quasi-singulated: a state where most of the parts which have been scattered onto a surface are sitting in a statically stable pose and not touching neighboring parts. The term "quasi" is used to indicate that most parts are singulated, but *NOT ALL* are.

and operate than traditional feeding solutions. Therefore, for such feeders to be useful and practical, a general architecture for their programming and control needs to be developed, implemented, and documented.

1.1 Architecture for Flexible Feeders

While on the surface it may seem that a standardized software control architecture for flexible feeders is an impossible task, the key to its success lies in the common functionality of the underlying components of all feeders. While the physical design of flexible feeders vary greatly, there are several sub-systems present in all designs whose functionality is the same. Therefore, by designing high-level software components to interact with the sub-system through a generic interface and server-level software components to encapsulate specific implementations of the sub-systems, a large portion of the code base need not change from feeder design to feeder design. Different server-level modules need to be programmed for each specific piece of hardware, but the high-level software objects may remain the same. The notion of feeder sub-systems was first introduced in the analysis of feeder throughput [1].

Consider, for example, that all feeders have some type of mechanism to present quasi-singulated parts to a sensing device. While this may take the form of conveyors, vibrational elements, or other devices, the end result (the sub-systems functionality) is the same: remove parts from bulk, singulate most of them, and present them to a sensor. By encapsulating (in a software object) the peculiarities of each type of part presentation device, they may be interchanged without affecting the rest of the control system.

Examining flexible feeders yields three distinct sub-systems which work together to feed parts. They are the parts presentation sub-system, the part pose determination sub-system, and the part retrieval sub-system, discussed below. Additionally, fundamental methods which must be included in the design of the generic interface to each sub-system are discussed.

1.1.1 Feeder Sub-Systems

1.1.1.1 Parts Presentation

The first major component of any flexible feeding system is the parts presentation sub-system. This is the component of the feeder which is responsible for

removing parts from the bulk supply and presenting them in a quasi-singulated fashion to the rest of the system.

There are two major modes of operation which may be identified for this sub-system: incremental and continuous. In the incremental mode, a higher-level object requests that the sub-system move more parts into the range of the sensing element. When the sub-system has accomplished this task, it signals the calling object that it has finished. The second mode of operation is continuous, in which parts are continuously moved past the sensor. In the continuous mode, a higher-level object first requests that the sub-system begin operation, and then waits for a signal that an additional batch of parts have moved into range. As long as the sub-system is operating, additional *new-parts-in-range* signals are generated. For example, the signals could be triggered by conveyors moving a certain range or a vibrational element operating for a certain amount of time at a certain amplitude.

Three key methods of a generic interface may be identified. The first is a method which moves more parts into the sensing area. The second two methods are applicable when the sub-system is in continuous mode. They are a method to start and stop the sub-system and a method to allow higher-level objects to register themselves to receive notification when an additional batch of parts are within the range of the sensor.

1.1.1.2 Part Pose Determination

The second major component of any flexible feeder is the part pose determination sub-system. This is the component of the feeder which is responsible for examining the parts which are within sensing range and determining which of those parts are in poses advantageous to retrieval and assembly. This sub-system is most often constructed using an industrial vision system, however other types of system could be envisioned (such as an array of photo detectors or an array of proximity sensors). In general, a vision system places the least amount of restriction on the types of parts that can be sensed.

The part pose determination sub-system can also operate in two modes. The first is to find a single retrievable part in the sensing area and the second is to find all retrievable parts in the sensing area. The first mode is safer at the expense of being

slower. If the part retrieval mechanism disturbs parts other than the one it is retrieving (such as when a robot misses a part-grab), then when the system finds the next part, it will find the part's current location (rather than its location before being disturbed). The second mode sacrifices safety in order to increase speed. Since only a single read of the sensor is performed for multiple parts being located, time is saved (with a vision system, grabbing the image can take as long as the processing of the image). In addition, when the parts presentation sub-system is operating in its continuous mode, the part pose sub-system must operate in the second mode. Since parts are continuously moving past the sensor, the sensor only gets a single opportunity to find each part.

Two key methods may be identified which must be provided in the generic interface to the sub-system. The first should return the location of a single part within the range of the sensor. The second method should return all the parts within the sensor's range.

1.1.1.3 *Part Retrieval*

The final component of any flexible feeder is the part retrieval sub-system. This is the component which is responsible for actually retrieving the parts which have been identified by the part pose sub-system. While this component of the feeder is most often an industrial robot, the fundamental requirement is a mechanism capable of reaching any location and orientation (usually a 2D area and single (vertical) axis of rotation) within a certain space. Industrial robots are generally used since they are well understood, well accepted, and often come with a programmable controller which can save substantial time in system development.

As with the previous two sub-systems, there are two different modes of operation of the part retrieval mechanism. This first, which corresponds with the incremental mode of the part presentation sub-system, is retrieval of a part from a stationary location. The part being retrieved is not moving and the retrieval mechanism simply needs to go to a position within its reach and get the part. The second mode of operation is retrieving the part *on-the-fly*. This mode corresponds with the continuous operating mode of the parts presentation sub-system and involves parts being retrieved from the presentation sub-system while the parts are still in motion. The retrieval mechanism

needs the ability to track a moving location and retrieve parts without the parts being at rest. Many commercial robot controllers provide a conveyor tracking mode to accomplish this function.

Two fundamental methods need to be provided which corresponds with the modes of operation. The first retrieves a part from a stationary location and the second retrieves a part from a moving location.

1.1.2 Philosophy

1.1.2.1 *Vertical Hierarchy*

The goal of the design of the controller system was to arrange levels of responsibility in a vertical hierarchy. Figure 1-1 illustrates. This technique has been successfully applied to previous control architectures at CWRU [2], [3]. At the high-level are the components of the controller which are concerned with the overall system, its current state, and interactions with the rest of the world. For example, is the system running or stopped? What part is currently being fed? or How many users are currently logged in?

At the mid-level are control components that are more specialized. They are concerned with their specific realm of responsibility. For example, the part retrieval subsystem is accountable for ensuring parts are retrieved, however, it is not aware of how those parts are singulated or located.

Below the mid-level components are the server-level components. They are responsible for encapsulating hardware-specific controllers and presenting a standard interface for those controllers to the mid-level components. This enables the mid-level controllers to perform their functions without knowing physically how that function is being accomplished. Consider the retriever again as an example (the retriever subsystem is tasked with retrieving parts). It uses the services of the part retrieval server to accomplish that goal. The specific physical hardware which is used to accomplish this task is encapsulated in the server, whether it is an industrial robot, a pick-and-place mechanism, or something else. This enables the hardware to be replaced without disruption to the rest of the controller; only the specific server must be re-implemented.

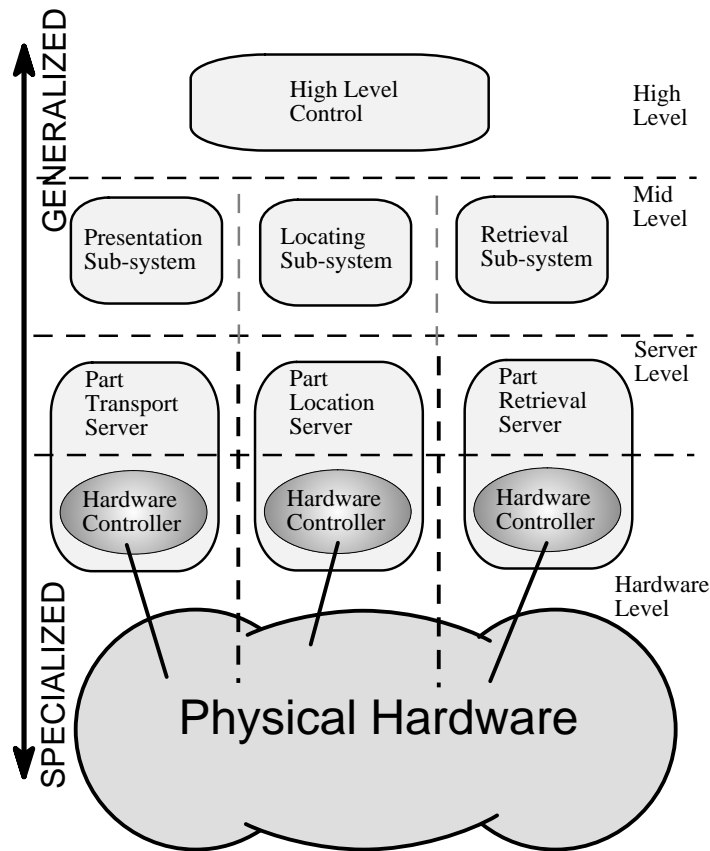


Figure 1-1: Generalized Control Architecture

Lastly, at the lowest level is the physical hardware and vendor-supplied proprietary controllers. At this level, each controller is responsible for physical control of the hardware. For example, the robot controller ensures that the robot does not *run away* or the vision frame grabber performs the proper timing and signaling to get an image from the camera. One goal has been to use off-the-shelf components to speed development time and to leverage each vendor's specific area of expertise. Depending on the specific hardware, there may be a small server program (in the specific hardware's programming language) running on the controller and responding to messages from the server-level component. In other instances, the hardware may simply respond to individual commands as they arrive.

1.1.2.2 Object Alignment

Complementing this vertical segregation of the controller, as one descends the hierarchy, is a progressively tightening alignment of control system objects with physical hardware.

At the high-level, the overall control is not concerned with (or even aware of) how the general goal of parts feeding is accomplished, beyond what is required for interaction with the mid-level components. This is shown on Figure 1-1 by the single box encompassing all objects at this level.

At the mid-level, components know their general tasks, e.g., *get parts from bulk* or *locate parts*, but they are not aware of the physical components used to accomplish those goals. There is some alignment at this point, but objects will sometimes use the services of components not directly related to their task. For example, the locator subsystem must know when to look for more parts. To determine this, it must use the services of the part transport server to know when there are more parts within its range of sensing. This is shown on Figure 1-1 by the light gray vertical lines.

At the server-level, the components are intimately aware of exactly what they are controlling. They are not concerned with other servers at this level, only with ensuring their own hardware component is accomplishing its task. This is shown by the dark black vertical lines.

This combination of vertical segregation with horizontal alignment serves to limit the amount of coupling in the components which make up the controller and, thus, help to localize any changes in control code that have to occur due to changes in physical hardware.

1.2 Previous Work

1.2.1 Physical Feeder Designs

The following section examines designs of flexible feeders and relates physical components of each design to the generalized sub-systems as described in Section 1.1.1.

Several manufacturers currently market flexible feeding solutions. Several other manufacturers have, in the past, marketed such solutions, but no longer do so.

Commercial feeders will be discussed first, followed by two CWRU feeder designs.

Images of commercial feeders, commercial software architectures, and commercial software GUI's in the following sections have been taken from web pages, brochures, and product manuals of the respective companies.

1.2.1.1 Adept

Adept Robotics, currently the leader in commercial flexible feeding technology, has marketed two feeder designs. The first is the Adept FlexFeeder 250 [4]. It uses two conveyors to present randomly-oriented, quasi-singulated parts to a vision system for pose determination. The two conveyors are mounted in-line, with the first slightly higher so as to drop parts onto the second. Part separation is accomplished by operating the second conveyor at a higher speed than the first. An Adept vision system is used to locate parts which are in favorable poses. An Adept robot is used to remove parts from the system. A third conveyor is used to move un-retrieved parts to an elevating bucket, which places the parts on the first conveyor for another trip through the system.

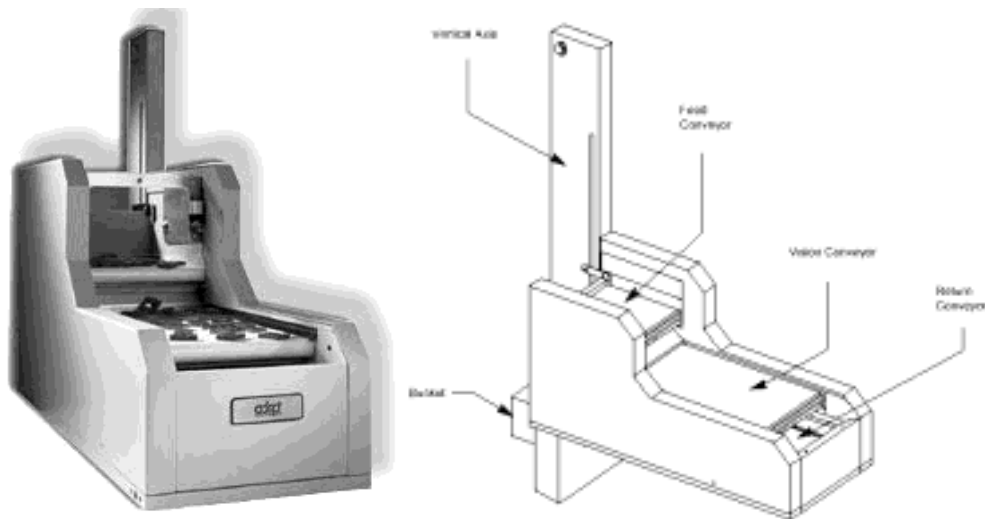


Figure 1-2: The Adept FlexFeeder 250

A second feeder currently being introduced by Adept is the iFeeder [5]. This feeder is similar to the FlexFeeder 250, but has one added feature. Incorporated into the second horizontal conveyor (the conveyor from which the robot retrieves parts) is a mechanism called a *popper*. It is a small voice-coil that travels (servo'ed in x and y) under the belt. It has the ability to hit from below (or *pop*) any part within its field of

motion. This action causes the part to tumble with the hope that it will come to rest in a new pose more advantageous to the subsequent assembly operation. To enable this feature, two vision systems are used. The first is used to determine parts which are not in the proper orientation and need to be re-oriented or *popped*. The second determines the pose of parts which have passed the popper and are entering the pick-up area.

Identifying the sub-systems of the FlexFeeder 250 is straight forward. The combination of the conveyors constitute the sub-system responsible for presenting parts; the Adept vision system is the part position determination sub-system; and the robot is the retrieval sub-system.

Identifying sub-systems of the iFeeder is not quite so obvious. However, one may apply the definitions of the sub-systems recursively to the feeder. On the first application of the definition, the robot can be identified as the retriever sub-system and the vision system which performs the final part location can be identified as the pose determination sub-systems. Next, the rest of the feeder's components (the conveyors, the first vision system, and the popper) can be grouped as the part presentation sub-system. Then the sub-system definitions can be applied to the presentation sub-system. This has the effect of defining a *mini-feeder* which, as a whole, can be considered the presentation sub-system.

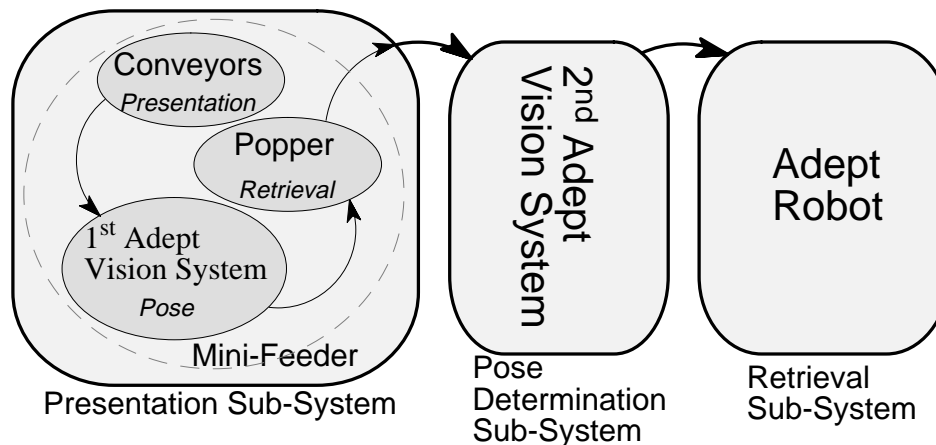


Figure 1-3: Adept iFeeder Sub-System Breakdown

The mini-feeder is composed of the conveyors, which can be identified as the presentation sub-system, the first vision system, which comprise the pose determination

sub-system, and the popper which is the retrieval sub-system. Figure 1-3 shows conceptually what is being defined.

1.2.1.2 *Seiko*

Seiko marketed a flexible feeder called the Smart Parts Feeder [6]. It consisted of an inclined, cleated conveyor to remove parts from bulk supply and a series of vibrating plates to move the parts through the system. There were two parallel, vibrating elements under two separate cameras which allowed parts to continue to move through half of the system while the robot retrieved parts from the other half of the system.

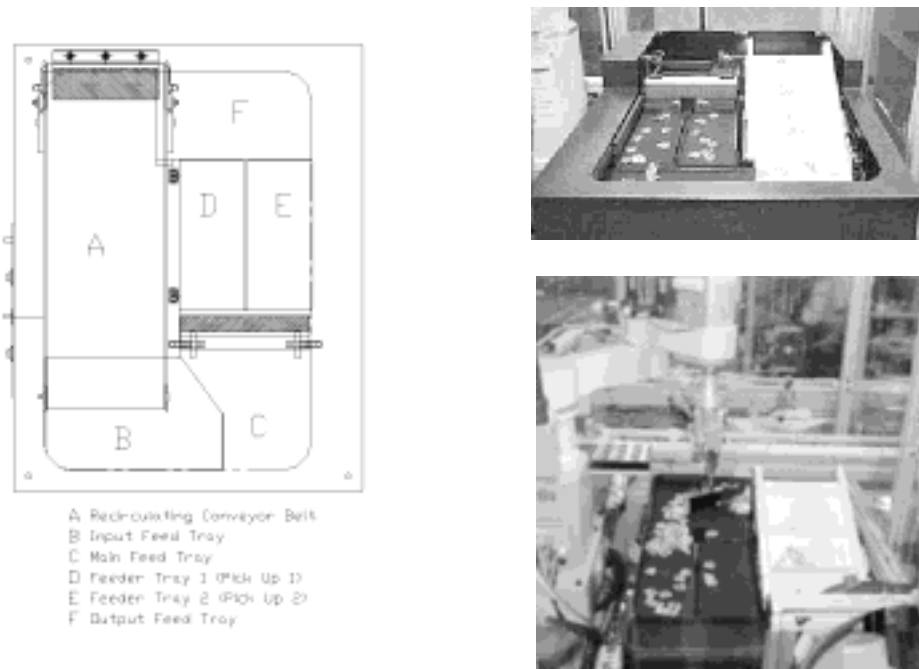


Figure 1-4: Seiko Feeder; Top Schematic and Views

The system, as delivered from Seiko, only included the series of conveyors and vibrational elements to move parts through the system. A robot and vision system from Seiko or another vendor would have to be purchased separately and integrated to construct a complete feeding system. Therefore the feeder itself would comprise only the presentation sub-system of a complete feeder.

1.2.1.3 Hoppmann Robotics

Hoppmann Robotics markets two feeders, the FC-2500 and the FW-4500 [7]. The FC-2500 uses an inclined, cleated conveyor to remove parts from bulk and an oval shaped conveyor to move parts under a vision system within reach of a robot. The FW-4500 uses two concentric rotating plates. Parts are dropped onto the inner plate, which is tilted relative to the outer plate. Parts slide from the first plate to the second. The second plate moves parts under a vision system for pose recognition and retrieval.

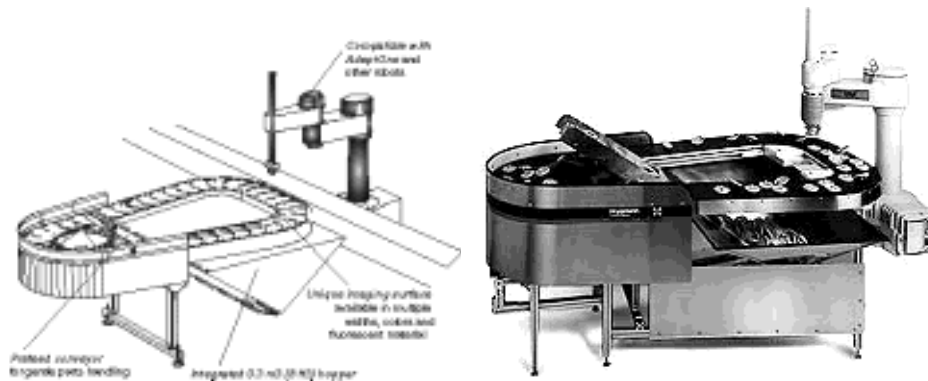


Figure 1-5: Hoppmann Feeder FC-2500: Schematic and View (with Adept Robot)

As with the Seiko feeder, no additional hardware, software, or controller is provided. The feeder is intended to interface with an existing robot and vision system and can therefore be considered only an implementation of the part presentation sub-system.

1.2.1.4 Intelligent Automation Systems

Intelligent Automation Systems produced a flexible feeder called the FPF-2000 [8]. The feeder used a bin with a trap door for holding bulk parts. Parts fell from the bin to a vibrating plate, which helped with singulation. Parts then would fall from the plate onto a conveyor which moved them past a vision system. A robot was used for retrieval. A second conveyor, coupled with an elevator/bucket, was used to return unused parts to the bin.

One can readily identify the trap door/vibrating plate/conveyors as the part presentation sub-system. The vision system was the part location sub-system and the robot was the part retrieval sub-system.

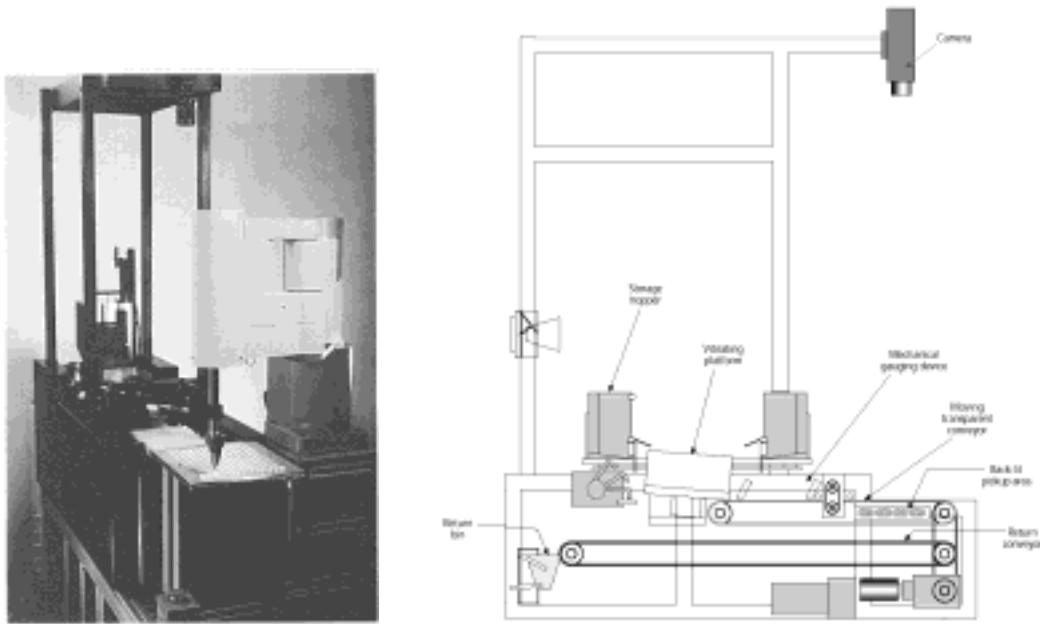


Figure 1-6: Intelligent Automation Systems: FPF-2000

1.2.1.5 *Robotic Production Methods Inc.*

Lastly, Robotic Production Methods Inc. (RPM) produced several flexible feeders [9]. They all used a combination of conveyors and/or vibrational elements to move parts through each system. One model (# 1540) used a cleated conveyor to lift parts from bulk. A series of vibrational elements then singulated and transported parts through a vision area. Unused parts fell back into the bin. A second model (# 1224) replaced the cleated conveyor with an additional vibrational element for removing parts from bulk. A third model (# 1040) used a V-block on a vibrational element to partially align cylindrical parts as they passed the vision area.

As with the Seiko and Hoppmann feeders, the RPM feeders were provided only as the conveyor/vibrational mechanisms which moved parts through each system. Therefore, the components as delivered from RPM would only constitute the part presentation sub-system. To create a complete feeder, one would have to add a vision system and robot from another vendor.

1.2.1.6 Original CWRU Feeder

The original CWRU feeder has been well documented in previous publications [10], [11], [12]; its design is briefly reviewed here. The feeder consists of three conveyors working in concert to singulate and present parts for retrieval. Figure 1-7 and Figure 1-8 show a CAD drawing and side-schematic of the original feeder. The first conveyor is inclined and is used to lift parts from a bulk hopper. By varying the angles of inclination and belt motion profile (acceleration, deceleration, and speed), a variety of parts are deliverable from the hopper. A 12" wide, 72" long conveyor is used. Parts slide down a ramp at the end of the inclined conveyor onto a horizontal conveyor. The horizontal conveyor terminates within the work envelope of a robot, where an underlit vision window is located. An 8" wide, 36" long conveyor is used. A third, fixed-speed conveyor (6" wide, 72" long) is used to return un-retrieved parts to the hopper.

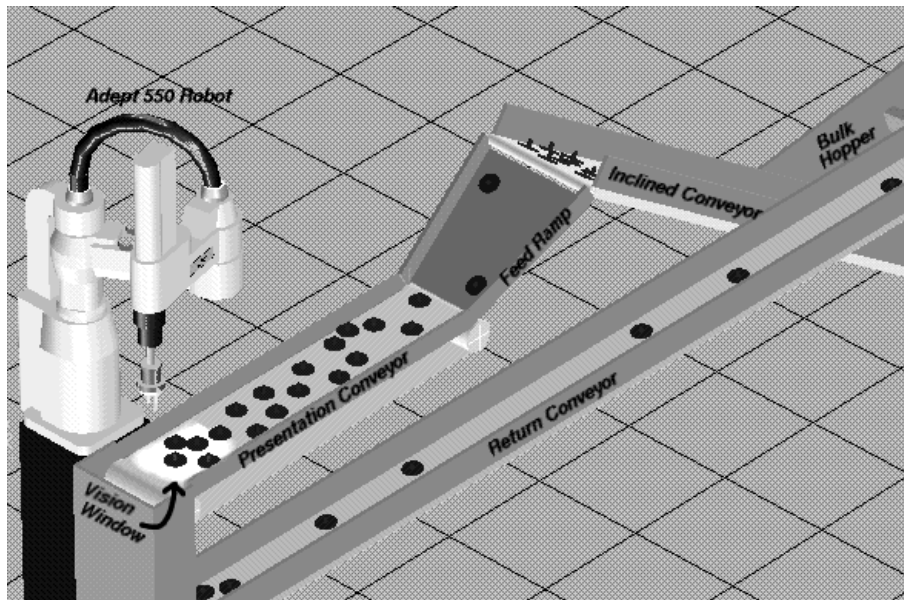


Figure 1-7: Original CWRU Feeder

The first two conveyors (inclined and horizontal) are under servo control. A stand-alone Galil 1500 motion controller is used to control and coordinate the motion of the two belts. Servo control is not required on the third (return) conveyor since it simply returns parts to the hopper, therefore a fixed speed motor is used.

Part pose is determined at the end of the second conveyor. An underlit window is machined in the conveyor and provides a back-lit, silhouette image to an overhead camera. An Adept vision system is used for part pose identification.

Parts which are in favorable poses (as identified by the vision system) are retrieved using an Adept 550 robot. The robot removes parts from the end of the horizontal conveyor.

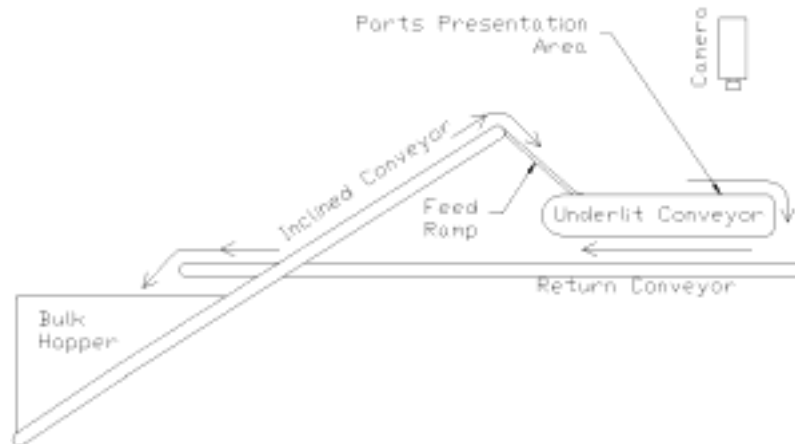


Figure 1-8: Side Schematic CWRU Original Feeder

The parts presentation sub-system consists of the three conveyors working together, the Adept vision system makes up the part location sub-system, and the Adept robot is the part retrieval sub-system.

1.2.1.7 New CWRU Feeder

The physical design of the new feeder, first introduced in [13], is similar to the original CWRU feeder. However, several changes were made in an effort to increase the throughput of the system as well as to enable the system to handle bigger parts. It consists of three conveyors working in concert to present quasi-singulated parts to a vision system and robot for pose identification and retrieval.

The inclined and horizontal conveyors are 24" wide and 48" long and are driven by servo motors. The return conveyor is 8" wide and 84" long and is driven by a fixed speed motor. The conveyors are mounted in a stand-alone frame, which also holds the robot, camera, and control system. Figure 1-9 shows a CAD view of the frame and conveyors (the robot, conveyor walls, and guarding are not included in this view). Parts

slide down a small ramp from the inclined conveyor to the horizontal conveyor. A fiber optic backlit is installed in the horizontal conveyor at the end near the ramp.

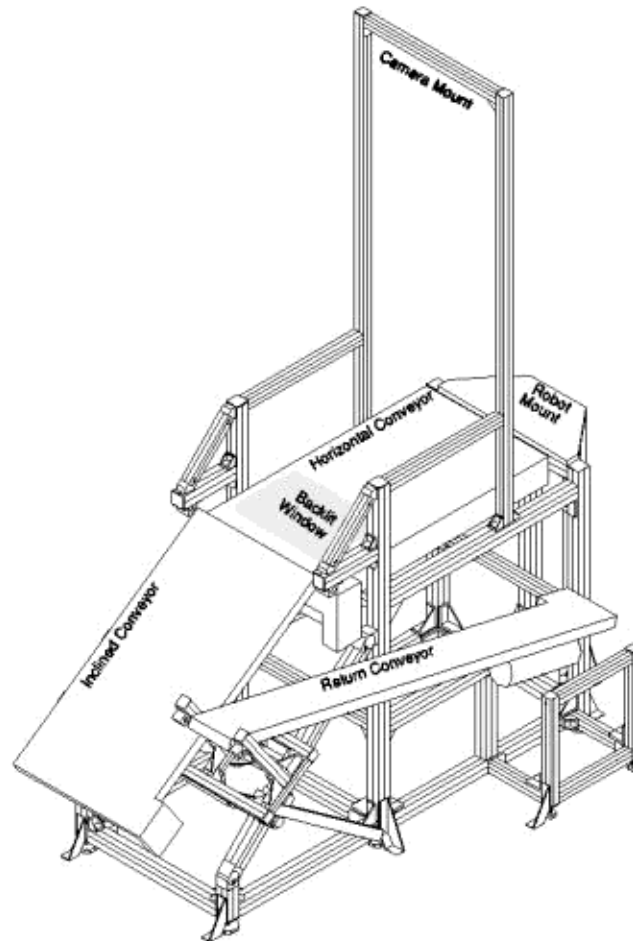


Figure 1-9: New CWRU Feeder: CAD View

A progressive scan CCD camera (Pulnix 6702) is mounted over the backlit window of the horizontal conveyor and is used for part pose determination. A PC-based Matrox vision system (Meteor II frame grabber and Matrox Imaging Library) is used for vision processing.

An Adept Cobra 600 robot is mounted at the end of the horizontal conveyor and is used for part retrieval. The robot is controlled by a stand-alone Adept AWC controller.

Summarizing, the conveyors are the presentation sub-system, the Matrox vision system is the pose determination sub-system, and the Adept robot is the retrieval sub-system.



Figure 1-10: New CWRU Feeder

1.2.2 Feeder Control Systems

A second component of existing feeder designs is the physical makeup of the control system. For example, how many controllers are used in the system? What is the physical location of those controllers? Over what channels do they communicate? (Local bus, serial port, Ethernet, IEEE 1394 FireWire, etc.)

1.2.2.1 *Adept*

The physical layout of the Adept system is the easiest to understand; a single controller running a single operating system is used to handle all aspects of the feeder. Internally, there are several physical boards connected via a VME bus; one board houses the main processor and is used for general OS activities and for servo-level control of the robot and feeder conveyors. A second board has a dedicated vision processor and is used for the vision system. Additional I/O boards may also be present. Because all processors reside on a common bus, communications is quick and handled by the OS.

1.2.2.2 *Intelligent Automation Systems*

The control system for the FPF-2000 is based on a PC for machine control and vision processing plus a stand-alone robot controller [14]. An I/O control card resides on the PC's local bus and interfaces with the hardware while a single-board vision

processor, also on the PC's local bus, is used for part location determination. A stand-alone controller is used to drive the robot and its associated accessories. A serial port connects the PC and external robot controller. Figure 1-11 illustrates the controller.

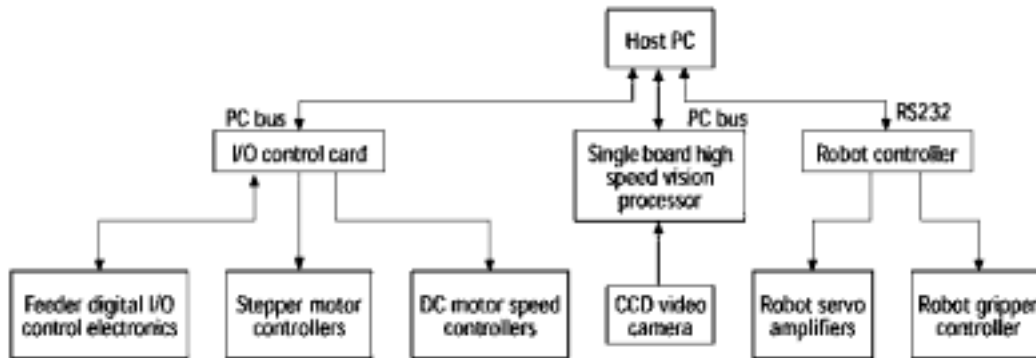


Figure 1-11: FPF-2000 System Controller Physical Layout

1.2.2.3 *Original CWRU Feeder*

Three separate control systems are used to operate the original CWRU feeder. An MVME167 single-board computer (SBC), running an RTOS (VxWorks), is used to oversee the operation of the system. A stand-alone Galil motion controller is used to drive the horizontal and inclined conveyors. The motion controller interfaces with the SBC via a serial link. Control of the robot and vision system is performed by an Adept MV controller. The Adept controller uses a VME backplane to enable communications between the main CPU (OS and robot control) and the vision processor (vision system). The Adept MV controller communicates with the SBC via a reflective memory network. This essentially allows the VME bus of the SBC and the VME bus of the Adept MV controller to share a segment of memory, which can then be used for message passing. Figure 1-12 shows an overall view of the control system.

1.2.2.4 *Others*

The other feeders discussed in the previous sections did not come with any formal controller, they were intended to be a part of a larger feeding system. Therefore, they usually provided a series of digital inputs and outputs for control. The overall controller would then toggle the input lines to cause the conveyors or vibrational elements to advance and watch the output lines for an indication of action completion.

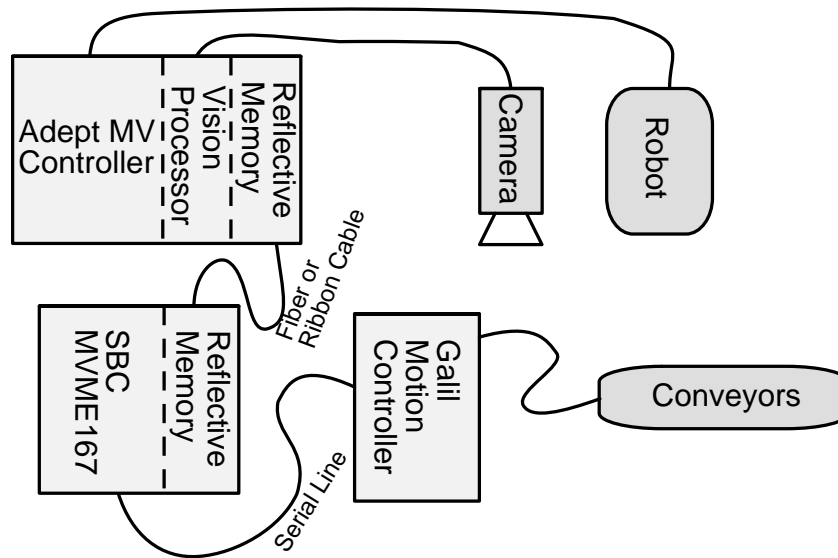


Figure 1-12: Original CWRU Feeder Controller Layout

1.2.3 Feeder Control Software

While the previous two sections examined feeder hardware and feeder control hardware, respectively, this section examines the software architecture used on previous feeders. The four solutions examined are Adept's feeder software, Intelligent Automation's control scheme, the application of Cimatrix's CODE to the problem of flexible feeding, and the software implementation of the original CWRU feeder. Unfortunately, exact details of how much of the functionality is implemented is considered proprietary by the manufacturers and not published or available for review.

1.2.3.1 Adept

The software for controlling the Adept FlexFeeder is written in V+ as an AIM module [15]. V+ is Adept's proprietary language; AIM is a front-end to V+ which enables the creation of graphical interfaces that are easy for plant floor personnel to use. Additionally, AIM enables easy programming of an Adept controller by shielding the end user from programming directly in V+. Internally, AIM uses a database to hold all the information necessary for an assembly operation. Boiler-plate V+ code is then executed using the information from the database to accomplish the required goals.[§]

[§]This is an overly simplified explanation of AIM. For complete details see the AIM documentation.

To setup the FlexFeeder for use, one uses the “FlexFeedWare Advisor”. This is a series of windows that asks questions specific to the current setup and guides the user through the steps necessary to get the feeder running. These include setting up a “FF 250 Module”, a vision task to find parts, a robot task to retrieve parts, and a specialized set of parameters tuned to the specific part currently being fed. Each of these steps creates a *module* which instructs the system how to accomplish that particular task.

For example, the “Part Tuning Advisor” ask a series of questions that are intended to help the user set up the initial parameters for feeding a part. Figure 1-13 shows one of the screens which asks questions of the user.

4) Part Setup Questionnaire

1 Select the Indexing Strategy

Info

- Index Vision Belt during cycle.
- Jiggle belt after index Cycle.
- Use the Jiggle feature alone to index parts forward.

2 Select Part Characteristics

Info

- Part is flat or does not roll.
- Part is round or rolls easily.

3 Bulk Storage Strategy

Info

- Use the Return Belt to hold most of the parts.
- Use the Feed Belt to hold most of the parts.

4 Return Bucket Configuration

Info

- Use standard Return Bucket strategy.
- Use the fastest Return Bucket strategy.
- Allow parts to slide off Return Bucket during dump.

< Prev Next > None 5 Do It

Figure 1-13: Example Q&A Screen of the “Part Tuning Advisor”

After the user sets up the feeder, a control panel is used to operate the system. A status panel allows one to watch the operation of the system as it functions. Lastly, a

“Feeder Record” panel is used to make fine-tuning the feeder parameters easy. Figure 1-14 shows the three panels.

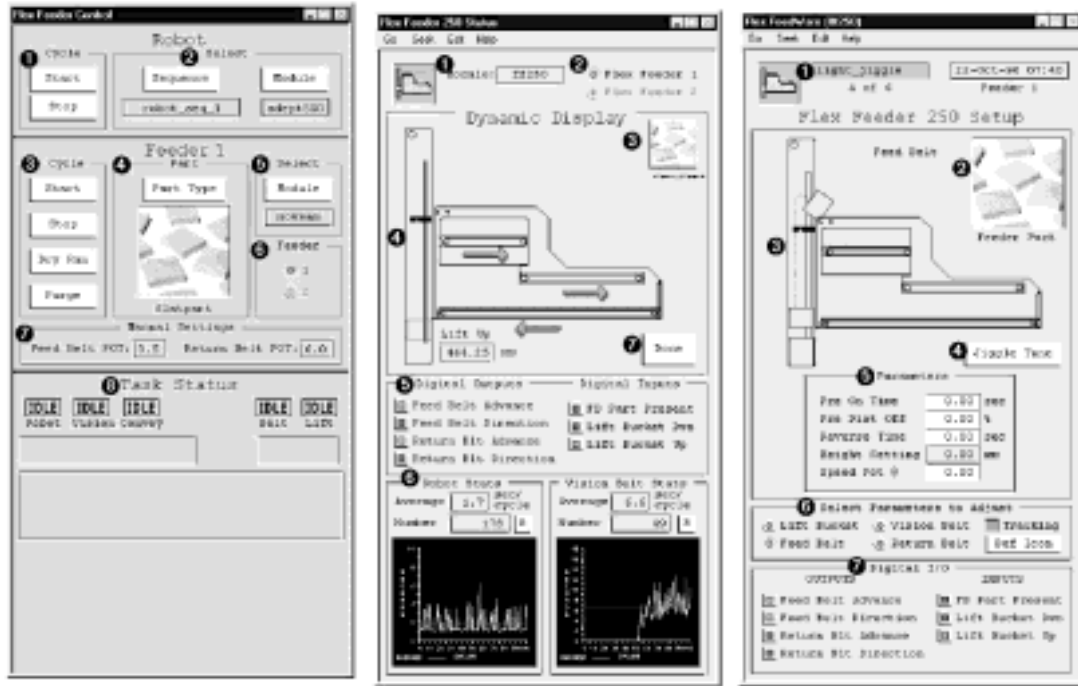


Figure 1-14: Control Panel, Status Panel, and Feeder Record Panel

While Adept has created a clean, helpful user interface, it is only applicable to a system using *their* feeder with *their* robot and *their* vision system. If one wanted to use a different robot or vision system, for example, their software would not be usable. Secondly, it is apparent when examining all the steps and parameters necessary for setting up a feeding task, that it is not a simple procedure with a small set of options. In addition, the ability of the system to adjust its parameters over time to improve throughput is not included. Lastly, because all their control boards reside on the same VME backplane, timing and communications is less of an issue; they are handled by the OS.

1.2.3.2 Intelligent Automation Systems

The control system for the FPF-2000 was operated by answering a series of questions designed to extract the necessary information to complete its operation [14]. The physical positioning of the mechanical elements were manually performed by the

operator (once set, they could be adjusted under computer control). Vision processing was performed using a pattern recognition strategy. The user would place an example part in the view of the camera and the system would *learn* the part's features and then be capable of locating additional parts as they passed the camera's field of view.

1.2.3.3 *Cimetrix*

While Cimetrix does not make a feeder, their open architecture machine control software, CODE [16], has been applied to the presentation sub-system from an Adept FlexFeeder 250 coupled with a Kawasaki JS5 robot and a Cognex vision system [17]. Using their software framework, an overall control task was created to oversee general system operation. Nodes (a software construct which "provide a hierarchical arrangement of coordinate systems"[17]) were used to convey location information between the vision system and the robot. Two worker threads were created to perform the actual vision work and part retrieval. The vision task would advance the conveyor a fixed distance and then locate all parts in the field of view. Part locations were placed into a queue. The second thread simply removed parts that were placed in the queue by the vision thread. Because the robot and vision system shared the same physical space, when no parts were in the queue, the robot would move out of the camera's field of view and wait for a part to be located.

1.2.3.4 *Original CWRU Feeder*

The control architecture for the original CWRU feeder was essentially a conglomeration of utility objects from the software architecture used to control the CWRU agile manufacturing workcell (documented elsewhere [2]), coupled with a single, monolithic master loop. While the software architecture was primitive as compared to a modular object-oriented design, it did operate the feeder reliably for extended periods and showed that a distributed control scheme, in which localized controllers handled time-critical tasks while a master controller orchestrated the bigger picture, was a viable operational strategy.

Three utility objects were used from the workcell control code: the DIO server, the robot server, and the conveyor server. The DIO server was used to read and change the state of the digital I/O points in the system. The robot server object was used solely

for its `robot::RunCmd(COMMAND_TO_RUN)` method, which allowed arbitrary V+ commands to be executed on the Adept MV controller. The conveyor server enabled easy control of the feeder conveyors and encapsulated all serial communication with the Galil controller.

Two threads were run on the SBC. The first thread interacted with the user through a text-based menu structure. It was used to initialize the system at start-up and to start and stop the system. The second thread ran the main control scheme which operated in a serial fashion. Initially, the system would try to find a part. If a part was not found, the system instructed the conveyors to advance more parts into the vision window and the part location routine called again. This continued until a part was found. When a part was located, the robot was then instructed to retrieve it. The system continued in this serial manner until the user chose to stop the feeder. Figure 1-15 shows a diagram of the three possible feeder states.

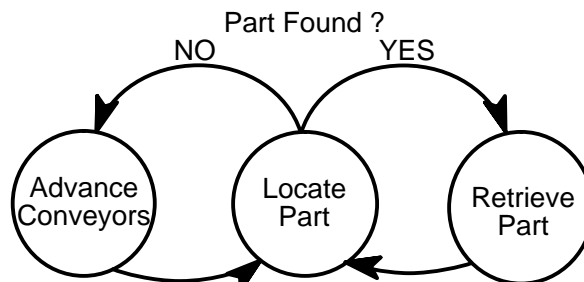


Figure 1-15: Original CWRU Feeder: System State Diagram

To cause the system to move more parts into the vision window, a function called `feed_advance()` was used. It encapsulated the algorithm for removing parts from bulk and presenting them to the vision system for pose determination. It used the conveyor server to accomplish its task. The algorithm used two different conveyor move profiles. The first, *big advance*, would cause both the inclined and horizontal conveyor to move. The horizontal conveyor would move a distance approximately equal to its length, the inclined conveyor a lesser distance. This action served to *load* the horizontal conveyor, or to fill its entire length with parts. The second motion profile was the *small advance*. This advance would only move the horizontal conveyor a short distance (approximately half the width of the vision window) and thus move a new section of the conveyor (and

hopefully more parts) into the vision window. This dual motion profile approach increased throughput over a simple, coupled index of both conveyors.

The vision system ran entirely on the Adept controller. The *back-door* `RunCmd()` method was used to execute the vision program (named “GO” and written entirely in V+) on the Adept controller. If a part was found, its location would be stored locally on the Adept controller. The part location program would then place a return value in reflective memory to indicate if a part was located (and to indicate what type of part was located).

The robot control was also performed completely from the Adept MV controller. However, in contrast to the vision system, which executed a complete program on the controller, the robot was controlled by simply issuing a series of V+ commands. Again, the `RunCmd()` method was used. To instruct the robot to retrieve a part, a sequence of V+ commands to cause the robot to move to the part location and then move to the drop location were sent. Control of the gripper was performed using the DIO server.

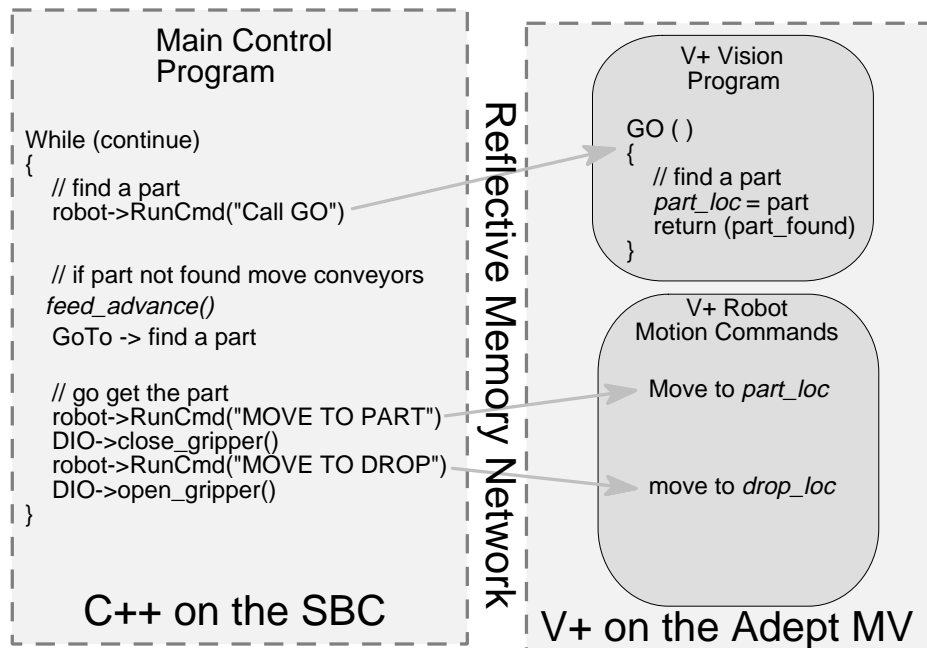


Figure 1-16: Original CWRU Feeder Control Scheme

To modify the robot’s behavior, one would have to rewrite the series of V+ commands which were sent to the Adept controller to affect robot motion. To introduce a new part into the system, in addition to modifying the code running on the SBC, a new

vision location program capable of finding the new parts would have to be written under V+.

1.2.4 Wrap-Up

Now that an examination of previous feeder designs and control strategies have been examined, it is important to consider the following points.

- All the feeders examined, while differing in physical form, contain the three major sub-systems defined in Section 1.1.1. It is this identification of sub-systems that enables a generalized software framework to be developed.
- There is a variety of configurations used in the physical layout of the hardware control of each feeder. Adept uses a single controller to perform all feeder functions; Intelligent Automation Systems and the original CWRU feeder use a more distributed approach; others provide no provision for hardware control beyond several digital I/O points for interfacing with another system. The important point is that the hardware for the control of each sub-system in all cases may be encapsulated behind a generic interface which will insulate any higher-level code from changes and nuances of any lower-level changes.
- There is truly a wide variety of software designs used for the feeders. If a manufacturing facility was to utilize feeders from several manufacturers, they would have to maintain the knowledge base to service and program each unique feeder. The situation is even more difficult for a system integrator who might want the ability to choose the best feeder for each unique project it faces.
- The philosophy of vertically segregating the software into levels of control, coupled with constantly narrowing realms of concern as one descends the hierarchy, is a valid approach to feeder control and would be applicable to all the feeders examined in this section.

1.2.5 Other Documentation Sources

In addition to this document, several other sources of information regarding the feeder control software have been produced. The documentation generation program Doxygen [18] was used to create a reference manual for the software [19]. Additionally, the source code itself is also bound into a publication [20].

1.3 Thesis Organization / Outline

The general outline and flow of the thesis is introduced and discussed below.

Introduction

This chapter has endeavored to first show the need for new feeding technology which is capable of meeting the requirements of the new manufacturing paradigm. Next, vision-based, flexible feeders are introduced as a means of accomplishing the requirements, however, they are currently difficult to program and thus a generalized software architecture applicable to all feeders would be of benefit. The philosophy of the architecture was introduced and its enabling concept, the identification of common sub-systems, was shown. Lastly, a selection of flexible feeders previously developed was discussed and shown to fit within the proposed architecture.

New CWRU Feeder: Control Hardware

This chapter will discuss the physical layout of the hardware used for the control of the new CWRU feeder. While the architecture of the control software is the main thrust of this thesis, a good understanding of the hardware being used for control (and being controlled) is important. And although, by design, the hardware does not impact the higher levels of control, it is very important to the server-level components as they must directly interface with (and encapsulate) the control hardware.

Overall System Architecture

This chapter gives a bird's-eye view of the software architecture. It is important to see the *big picture* before delving into the details of any particular section of the software. The major levels of control are identified: high-level, mid-level, server-level, and hardware-level.

High-Level Components

This chapter will examine the objects which comprise the high-level (those concerned with overall control and coordination and user interface/interaction). This includes the `main_control` object and the infrastructure necessary to enable multiple remote users to interact with the system. Additionally, the user GUI will be examined.

Mid-Level Components

This chapter will examine the components which make up the mid-level objects. Four asynchronous threads are used to accomplish feeding goals. The first three are aligned with the major feeder sub-systems. The fourth provides a framework for implementing an intelligent feeder-parameter-adjusting mechanism, which could be used to allow on-line fine-tuning or optimization of feeder throughput. The class factory design pattern is used to construct part factories which enables one to change feeding scenarios on-line without system shutdown through the use of part specific objects. Two additional utility objects, the conveyor encoder and the parts queue, are also discussed.

Server-Level Components

This chapter will discuss the server-level server objects which are used to wrap the hardware which they control and present a generic interface to the rest of the system. Server objects include a robot motion server, a robot position server, a vision server, a conveyor server, and a digital I/O server.

Hardware-Level Components

This chapter shows how each physical controller in the system is utilized. Hardware controllers include the Adept AWE controller, the Galil motion control card, and the Matrox Meteor II frame grabber and Matrox Imaging Library (MIL).

Results, Future Work, Conclusions

This chapter will first examine the results of running the feeder with various parts and feeding scenarios. It will also provide some closure to the discussion of the software architecture. Finally, an abundance of future work opportunities are presented.

2 Feeder Control Hardware

This chapter endeavors to describe the physical hardware used in the new CWRU feeder. While the physical hardware is not important to the upper levels of the control architecture, it is intimately tied to the server-level objects. The server-level objects interface directly with the physical controllers and serve to encapsulate the peculiarities of the specific hardware. It is also important to examine the hardware in light of how the various controllers are connected; the physical layout and interconnection dictates how communications between the controllers are performed. There are also two instances in the system where timing requirements dictate that the controllers share physical signal lines. It is important to understand where the tight timing is required so that it may be accounted for when replacing hardware in the system or constructing a new feeder using different hardware.

2.1 Overall Control

The overall control and coordination of the feeder is performed by a standard desktop PC running Microsoft Windows NT 4.0. The system uses a Pentium III processor at 600 MHz and has 128Mb of RAM. The system is programmed in C++ (using MS Visual C++ 6.0) and communicates with the rest of the system controllers using several techniques.

User interaction with the system is accomplished through the use of a networked user interface. The interface is written in Java for portability and runs on a variety of hardware/software platforms. Communications occurs over Ethernet using standard sockets. From a control standpoint, no code is executed from the user interface; all high-level control code executes on the host PC. The communications are also written such that network delays (or a slow computer hosting the user interface) do not cause the throughput of the system to suffer. Therefore, one can consider the physical control hardware ending at the host PC. The platform on which the user interface is running simply conveys user requests and displays the state of the system, it is not involved with system control.

2.2 Vendor Supplied Hardware Controllers

As mentioned in the introduction, one goal of the controller design has been to use, as much as possible, vendor supplied, off-the-shelf controllers. By using the controllers supplied by each vendor with their hardware, each vendor's expertise may be leveraged in the development of the feeder controller. It is unwise to spend a lot of time developing, implementing, and testing control code when the vendor has already performed that work and provides the solution with the physical hardware. For example, while one could buy an Adept robot and re-write the entire servo level library, it would be foolish to do so. Adept has built its reputation on the physical control of robots and that expertise should, whenever possible, be utilized to its fullest potential. It is much wiser to write a small wrapper object that presents a generic interface to the rest of the system and converts generic calls to Adept-specific instructions than to re-write then entire servo-level control code for the robot.

There are three hardware controllers in the system. A stand-alone Adept AWC controller is used to drive the robot, a Galil 1822 motion control card is used to drive the conveyors, and a Matrox Meteor II frame grabber is used to drive the camera and acquire images.

2.2.1 Adept AWC Controller

The Adept Cobra 600 robot is controlled using the Adept AWC controller [21]. The AWC controller is a stand-alone, headless controller which uses an Ethernet connection to remote display its text-based user interface on a PC running a special terminal program. The terminal program, called AdeptWindows [22], allows full control and programming of the system from any networked PC. The controller boots from a local flash disk and then accesses a remote hard drive via NFS (the hard drive is located on the PC and is exported using Sun's Solstice PC NFS software). Program storage then occurs, physically, on the PC's hard drive, which enables easy backup of control code. Programming the Adept system is accomplished via the remote terminal using a text editor built into the OS (Adept's *SEE* editor) or via an off-line editor which runs under Windows.

All communications between the Adept controller and the main control (running on the PC) occurs over TCP/IP sockets. This enables the control PC and the Adept controller to be physically located anywhere. In this setup, both controllers are connected to a common Ethernet hub. The hub also connects the feeder to the rest of the Internet.

In addition to controlling the robot, the AWC controller also provides a generous quantity of digital inputs and outputs. The I/O points connect to the controller through several Adept-supplied patch cables. The free end of the patch cables may then be attached to a termination strip for easy connection to the rest of the world. In total, there are 44 inputs and 40 outputs.

The final input to the Adept controller is the encoder from the horizontal conveyor. The controller needs to observe the encoder if the robot is to track the conveyor for part retrieval. This is an important point; if the controller and robot are replaced with a different vendor's equipment, then one must ensure that the proper hardware lines are connected such that the robot may track a moving conveyor. Conveyor tracking was one of the fundamental methods required of the retrieval sub-system as discussed in Section 1.1.1.3.

2.2.2 Galil 1822

Control of the inclined and horizontal conveyors is performed using a Galil 1822 motion controller [23]. The controller interfaces with the PC via the PC's PCI bus. Communications with the card is performed using Galil-provided library functions. It is interesting to contrast this controller with the stand-alone Galil controller used on the original CWRU feeder. Since both controllers use the same simple command language, porting the conveyor server from the version of the original feeder (running under an RTOS communicating with a stand-alone controller) to the PC-based version used on the new feeder (running under Windows NT with a PCI controller) was accomplished in a couple days time. The private `send()` and `receive()` methods of the conveyor server object were altered from serial communications to using the Galil communication library functions. Everything else remained the same. This also illustrates the utility of

leveraging vendor-supplied code. By using Galil's library functions, significant time was saved in porting the code.

To interface all the necessary signals to control the conveyors (11 wires per conveyor), a break-out box (Galil ICM-2900 Interconnect Module) was used to provide easy physical connections. A 100 pin SCSI style connector with cable (also vendor supplied) was used to connect the PCI card and the break-out box.

In addition to controlling the conveyors, the Galil controller also provides several digital input and output points (8 input and 8 output). The physical interface with the I/O points is also provided through the break-out box.

Lastly, the final connection to the Galil controller is the trigger signal used to start an image grab (discussed in the following section). This signal is connected to a special *latch* input which records the value of the conveyor encoder within 25 μ s of the signal transition. It is important to have this capability to enable a tight coordination between the vision system and the conveyor encoder. Section 2.3 below discusses this connection and the encoder connection into the Adept controller in detail.

2.2.3 Matrox Meteor II

A Matrox Meteor II frame grabber [24] is used to obtain images from the CCD camera. The frame grabber interfaces with the PC via the PC's PCI bus. Communications are accomplished using Matrox Imaging Library (MIL) functions [25]. The MIL takes care of setting up the system for use and ensuring that proper timings and protocols are observed. Using the MIL is another example of leveraging vendor-supplied code to speed system development. The frame grabber itself performs all the necessary signaling to acquire an image from the camera. The camera operates in an asynchronous reset mode, in which an image is only grabbed when the frame grabber requests it. To request an image, the frame grabber pulls a signal line low for a set period of time. That duration is the amount of time the CCD element collects light; the equivalent of an electronic shutter. After the line returns to a high state, the image is sent from the camera to the frame grabber.

The frame grabber board provides several programmable timers which are used to generate the necessary timings for acquiring an image. By manipulating the timings, one may make on-the-fly changes to the exposure setting of the camera.

In addition to connecting the image acquisition trigger to the camera, it is also connected to the Galil controller. This is the signal which causes the current conveyor encoder value to be recorded. By driving both the camera and conveyor encoder latch simultaneously, tight coordination between the conveyor encoder and the vision system is ensured.

2.3 Time-Critical Connections

While most of the communication between system components occurs over standard channels, there are two situations which must occur with very tight timing. They are conveyor tracking and image acquisition/conveyor encoder reading.

2.3.1 Horizontal Conveyor Encoder

To enable the robot to accurately track the horizontal conveyor for part retrieval, it is necessary for the Adept controller to have access to the encoder signal from the conveyor's motor. This is accomplished by splitting the conveyor encoder signal lines. One set of lines is sent to the Galil controller so that it may maintain servo-level control of the motor while the second set is sent to the Adept controller. A total of 6 signal lines (3 twisted pairs) need to be reproduced; two differential signal lines for the quadrature encoder and a differential signal line for the sync signal. Power to the encoder is supplied only from the Galil controller.

A calibration procedure is performed between the conveyor and the Adept controller so that a mathematical relationship is established between the output of the conveyor encoder and physical motion in the robot's workspace. This enables the robot to move at the correct speed relative to the conveyor no matter what the conveyor is doing; e.g. constant speed, accelerating, or decelerating. This allows the speed of the conveyor to be adjusted at any time while still ensuring that the robot will accurately track it. It is expected that the speed of the horizontal conveyor will be a critical parameter that can be adjusted to optimize throughput (such adjustment would be performed by an auto-adjusting object).

The last issue to resolve is keeping the encoder values as seen by the conveyor controller and the robot controller in synchronization. In general, the two encoder inputs might be implemented using different size counters. In this specific case, the Galil encoder input uses a 32-bit signed number, while the Adept encoder input uses a 24-bit signed number. Care must therefore be taken in recording and accounting for the roll-over in each counter. In practice, the Adept encoder rolls over every 5 to 15 minutes; the Galil encoder every 1 to 2 days. A conveyor encoder object (discussed in Section 5.3.1) is used to maintain a consistent view of both encoders.

2.3.2 Camera Trigger / Conveyor Latch

The second area in which tight timing is required is in the synchronization between capturing an image and recording the value of the conveyor encoder. This is critical because the location of the parts in the robot's coordinate frame (as determined by the vision system) is directly dependent on the value of the conveyor encoder when the image was acquired. If an errant value for the conveyor encoder was used in calculating a part location, the robot would be unable to successfully retrieve the part.

To ensure this value is not miscalculated, the signal line which is used to initiate an image grab on the camera is also mirrored into the conveyor controller and used to trigger a hardware latch. The hardware latch (once properly initialized) will record the value of the conveyor encoder within 25 μ s of the latch's being triggered. That value may later be retrieved and the latch reset. This is accomplished using the conveyor encoder object discussed in Section 5.3.1.

While it is additional work to set up the extra signal line to the Galil controller and initialize the hardware latch, it is necessary. One can not simply call `vision→grab_image()` followed by `conveyor→get_encoder()` on successive lines of code. The call to `grab_image()` could potentially take between 25 and 250 milliseconds while the call to `get_encoder()` could also take that much time. Also considering that Windows NT is a preemptive multi-tasking OS, it is conceivable that the entire task could be swapped out for an indeterminate amount of time at any point. Consider that if the conveyor is moving at 100 millimeters per second, a delay of even 250 millisecond between image capture and encoder reading would cause the robot to miss the true part

location by 25 millimeters! It then becomes obvious that the two events must be tightly coupled via hardware signals and not left to the expected (and often falsely assumed) speediness of the computer.

2.4 Control Hardware Layout Diagram

The physical layout of the hardware is shown in Figure 2-1, which includes all critical connections. The large gray box on the left shows the PC used for overall control while the large gray box on the right shows the Adept AWC controller. Inside the PC is the Matrox Meteor II frame grabber and the Galil 1822 motion control card. An Ethernet controller is also in the PC. Inside the Adept controller is the robot controller, the I/O module, and the input for the conveyor encoder (in reality, some of this functionality is on a single card). The Adept controller also includes an Ethernet adaptor. An Ethernet hub connects the PC and the Adept controller to one another and to the Internet. Remote users can be located anywhere a TCP/IP connection is available.

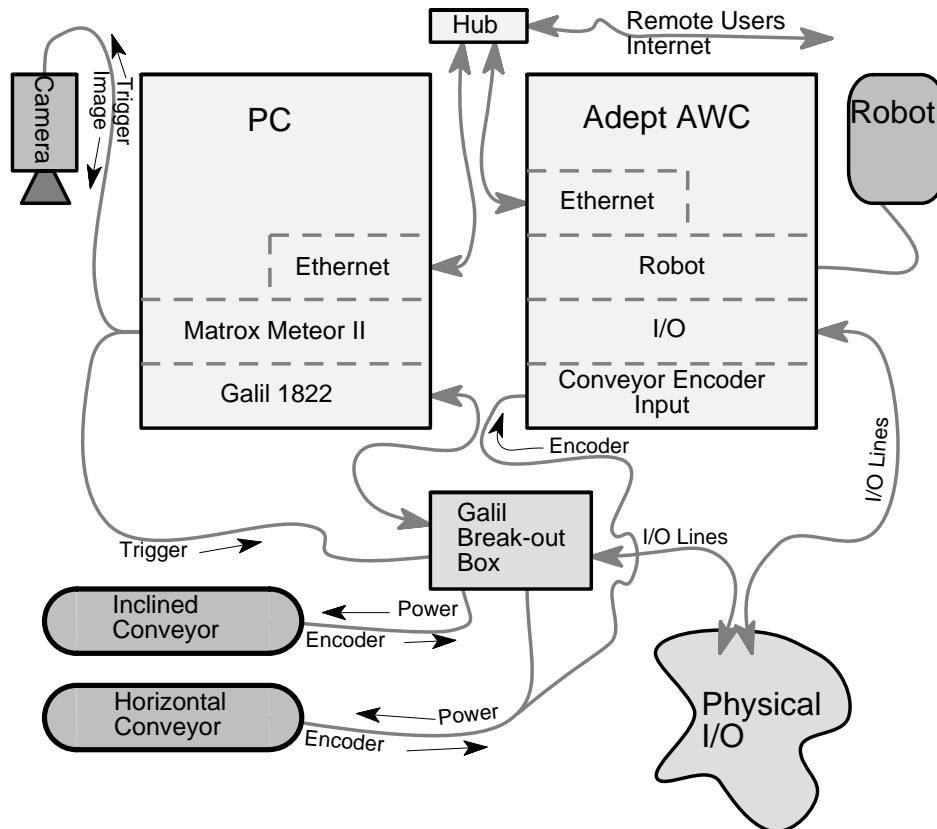


Figure 2-1: Layout of the Feeder's Control Hardware

A break-out box is attached to the Galil control card and provides easy connections for the conveyors and I/O. The horizontal conveyor encoder attaches to both the Galil break-out box and the Adept controller. The camera is attached to the Matrox frame grabber. The trigger line used to initiate a frame grab is mirrored into the Galil break-out for conveyor encoder latching. Lastly, the robot attaches directly to the Adept controller. Connection to the Adept I/O lines is provided by several Adept supplied cables.

3 Overall System Architecture

When describing the software architecture, it is helpful to begin with the big picture. Then, in later chapters, as specific objects and interactions are discussed, it will be much easier to understand how that particular component fits into the whole. This chapter, therefore, presents an overview of the complete software architecture. The discussion starts at the high-level and proceeds down to the hardware-level.

In keeping with the design philosophy of the controller, an architecture was developed which uses a multi-level segregation of objects which become progressively more specialized as one descends the hierarchy. Figure 3-1 shows an overview of the major system components.

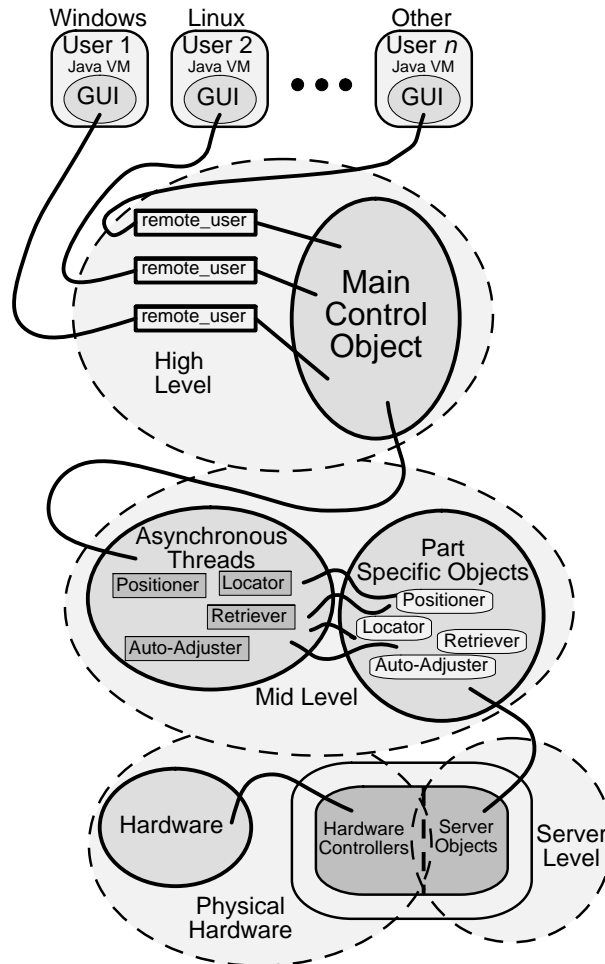


Figure 3-1: Software Architecture Overview

3.1 High-Level

At the highest level are remote users. Multiple users may log into the system and monitor its operation. One user who has *control* may send requests to the feeder. Each user interacts with the system through a GUI (Figure 4-1) which runs remotely on the user's desktop CPU. The GUI is written in Java to enhance portability. When a user logs in, a `remote_user` object is created on the feeder controller which interacts with the remote GUI. Communications occur over a TCP/IP socket.

A `main_control` object serves as the high-level controller. It is responsible for generalized functions, such as starting and stopping the feeder in an orderly manner, maintaining a list of the currently logged in users, and maintaining the objects that reflect the current state of the system.

The `main_control` object with all `remote_user` objects constitute the high-level control objects as depicted in Figure 1-1.

3.2 Mid-Level

Below the `main_control` object are four independent, asynchronous threads. Three of the threads are used to accomplish the goals of the major sub-systems of the feeder; the fourth is used for system optimization. The first is a presentation thread that is responsible for getting parts from bulk to a quasi-singulated state. The second is the locator thread whose job is to locate parts that have been singulated by the presentation thread. The third thread is responsible for retrieving parts that have been located by the locator thread. The final thread, which does not have a direct relationship to a feeder sub-system, enables the implementation of an auto-adjusting function. This thread allows on-the-fly adjustment of any feeder parameter dynamically during operation. By using various sources of data (such as current feeder parameter settings, feeder state, current and past throughput, or feeder models) the thread may make intelligent adjustments to the feeder to stabilize or optimize its throughput.

While the threads are used to accomplish the goals for each feeder sub-system, they do not contain any code specific to a given part. Instead, they each utilize a corresponding object, called a *part specific object* or PSO, specific to the current part to

accomplish their tasks. There are four such separate objects defined: a part presenter, a part locator, a part retriever, and a part auto-adjuster. Each object is derived from an abstract base class and is constructed using the class factory design pattern [26]. The `main_control` object, depending on the part type selected by the user, determines the specific factory to instantiate, creates the PSOs and passes pointers to the asynchronous threads. By only accessing the base class methods, the threads may use the services of the objects without knowing the part type for which the object is specifically designed. This enables the part currently being fed to be changed on-the-fly by simply changing the current state of the feeder to stopped, selecting a different part, instantiating new PSOs, then changing the state back to running.

The PSOs encapsulate all the details which are required for any particular feeding situation. To add a new part to the feeder, one only needs to create new PSOs (inherited from the abstract classes); no other code needs be altered.

The combination of the independent threads working in tandem with the PSOs comprise the mid-level objects as presented in Figure 1-1.

3.3 Server-Level

Operating below the independent threads and PSOs are the server-level components. In the controller there are 5 server-level objects. They include an object for controlling the inclined and horizontal conveyors, an object for controlling the vision system, an object for controlling the robot, an object for reporting the instantaneous position of the robot, and an object for changing and reporting the state of the system's digital I/O points. Each server is constructed using the singleton design pattern [26b] with the double-check pattern modification [27]. This allows the PSOs to get instances of the servers as they need them, without regard to each server's current state. It is also important to only have a single instance of each server in the system because they directly interface with physical hardware. The double-check modification ensures that only one object is ever created, even when multiple threads call the object's `instance()` method simultaneously.

Four of the servers (the conveyor server, the vision server, the robot server, and the robot position server) directly map to physical hardware. The digital I/O server does

not directly map to a hardware component since the I/O points of the system are physically located on the conveyor controller and the robot controller. The I/O server encapsulates the physical location of the I/O and presents a consistent numbering scheme to the rest of the software. To accomplish this, it uses the conveyor server to change the state of I/O on that controller, while it directly interfaces with the Adept robot controller to change its I/O.

Each server encapsulates the specific conventions of the hardware with which it interacts and presents a generic interface to the rest of the system. This allows any piece of hardware to be replaced with a similar component without effecting the rest of the control code. All changes to accommodate the new hardware would be contained within the new server. Replacing the robot with one that is faster or more capable would be an example.

An additional use of the servers is to regulate access to the hardware. While software is often written to be re-entrant, real hardware can only do one thing at a time. Therefore, it is vital to ensure that conflicting commands are not issued to the hardware simultaneously. Further, it is important that the hardware finish its current task before starting the next. Many of the functions of the hardware can be considered atomic; picking a part and placing it somewhere or grabbing an image and analyzing it are two examples. Improperly interleaving commands could cause problems or even damage to system hardware components. Monitor constructs [28] are used in the servers to ensure all operations are properly regulated. Incorporating this functionality into the servers allows mid-level objects to make requests of the servers without forcing the objects to coordinate with each other.

3.4 Hardware-Level

At the lowest level are the hardware controllers. They are responsible for the servo-level control of the physical system components. For example, the Adept controller ensures that the robot physically goes where the robot server has instructed and that it does not run away in the process. Using the controllers as provided by the vendors allows one to leverage each vendor's expertise in their particular areas. Additionally, the

entire system may be developed in less time by avoiding re-creating code which each vendor has already written and tested.

Three hardware controllers, as discussed in Chapter 2, are used in the feeder. Both local executing programs and single command requests are used when interfacing with the hardware controllers. Each controller is programmed or commanded using its native programming language.

3.4.1 Galil 1822

A combination of a local executing program and individual requests are used in interfacing with the Galil controller. Each request from the conveyor server arrives as an independent sequence of commands which will cause the desired movements. No local program is used for the conveyor server. However, the I/O server uses a local program on the controller to keep the current state of the input up to date. If an input point changes state, the program notices and causes an interrupt on the PC. This interrupt is relayed back to the I/O server which can then act upon it accordingly.

The Galil controller is programmed using its two-letter programming/command language [29]. For example, TP causes the controller to report the current position of the conveyors or RS causes the controller to reset itself. The conveyor server object encapsulates this language behind its generic interface.

3.4.2 Matrox Meteor II

The frame grabber is programmed through the MIL. The vision server object encapsulates the sequences of MIL function calls which are used to perform its requests. No program is run natively on the frame grabber itself.

3.4.3 Adept AWC

Four separate programs run on the Adept controller to enable it to service requests. The first program receives requests from the robot server and drives the robot. The second program is used to report the instantaneous position of the robot. A separate program is used here so that other control components may determine the position of the robot even if it is in the middle of performing a motion. The third program is used to change the state of output points, while the fourth program monitors the state of the input and notifies the I/O server if there are any changes.

Each program on the controller uses a TCP/IP socket to communicate with the appropriate server object running on the PC. Programs on the Adept controller create a listening socket and wait for the respective server objects on the PC side to initiate the connection.

4 High-Level Components

There are several major components which make up the high-level control. They are the remote user interfaces, the local `remote_user` objects and the `main_control` object. Each will be discussed in the following sections.

4.1 Remote User Interface (GUI)

The remote user interface allows one to control the feeder from anywhere there is a TCP/IP connection and a Java VM. It also enables multiple, simultaneous users to log into the feeder at one time. A rudimentary user ID/password scheme is used to control access. While multiple users may be logged in, only a single user may have *control* of the feeder; all other users may only observe the current system state and statistics.

4.1.1 Front-End

The front-end is composed of four major areas: an area for logging into the feeder controller, a control area, a system state area, and a user input area. Figure 4-1 shows an overview of the control panel.



Figure 4-1: Feeder GUI

The first area is used to log into the system. It accepts a username and password and has a button which sends a login request. The second area allows the user to get control of the system. The area contains an icon indicating current control allocation (yellow→available, green→control granted, red→someone else has control) and a button for getting or releasing control. In addition to the color of the icon, the text on the button also reflect the current control state. Figure 4-2 shows the area when the current user has control and when someone else has control.



Figure 4-2: User Has Control (left) and Someone Else Has Control (right)

The third section shows the status of the system. There is a series of icons which indicate the current state of the system: STOPPED, PAUSED, or RUNNING. A text box indicates the current part scenario being fed (this could be a single part or a combination of parts being fed simultaneously). A second text box indicates the part most recently fed (this can change when feeding multiple parts at once). Additional text boxes indicate the total number of parts fed, the current feed rate (since the current feeding scenario began), the run time for the current feeding scenario, the system up time, and finally the total number of users logged in. Figure 4-3 shows the system status area while running.

The final area allows a user who has control to operate the system. The drop down provides a menu of possible parts (or scenarios) to feed. Next are three entry box/slider combinations which allow the speed of the horizontal conveyor, the inclined conveyor and robot to be set. The speed of the horizontal conveyor is specified in millimeters per second; the speed of the inclined conveyor is specified as a percentage (0% — 200%) of the speed of the horizontal conveyor. This allows one to set the ratio of the conveyor speeds (important in ensuring part separation) and then speed up or slow down both conveyors at once (by setting the horizontal speed). Lastly, the speed of the

robot is specified as a percentage (0% — 200%) of maximum speed.[§] The final input area contains four buttons to start and stop the feeder, to pause the feeder, and to enable the auto-adjusting mode of the feeder. When the auto-adjusting mode is enabled, most of the user input area is disabled and the system status area is updated to indicate this state. Figure 4-4 shows the user input area. It also shows the system state area when auto-adjusting is enabled and not enabled.



Figure 4-3: System Status Area

Lastly, an additional window may be brought up which shows the messages being passed between the interface and the feeder controller (Figure 4-5). This is helpful in system debugging and shows the amount of traffic between the two systems. Messages may be displayed in raw form (as passed) or in human readable form (both forms are shown in the figure).

[§]“Maximum speed” is dependent on the specific robot and controller currently being used. Speeds over 100% move the robot faster at the sacrifice of accuracy and controllability. Speeds over 100% are generally not linearly related, i.e. 200% is not, physically, twice as fast as 100% for Adept robots.

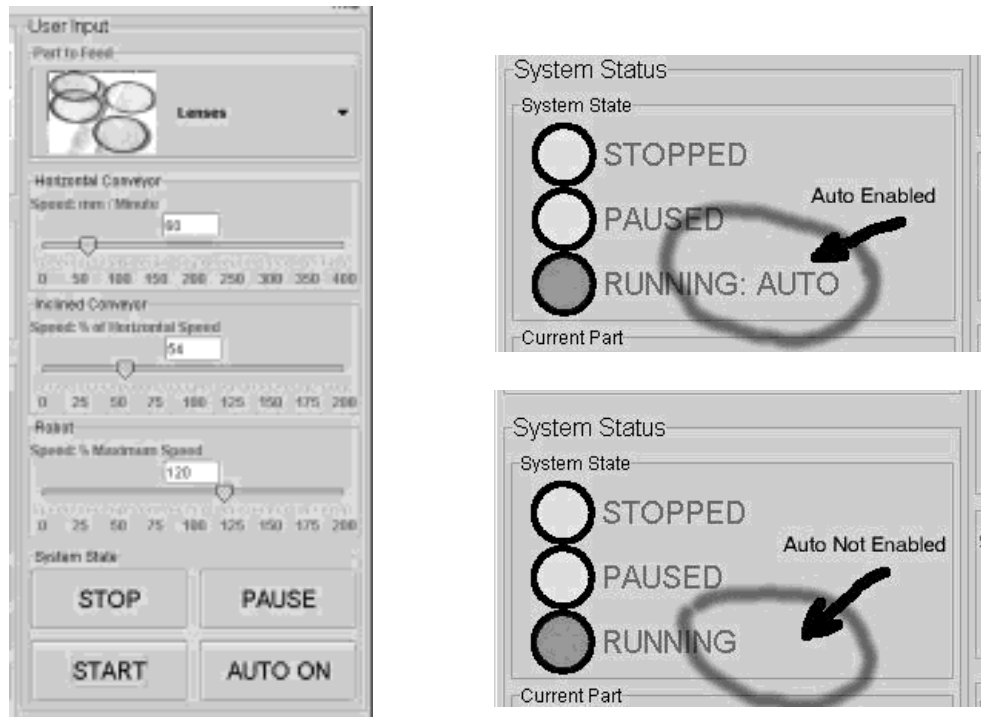


Figure 4-4: User Input Area and Auto Mode Indicator



Figure 4-5: Message Window

4.1.2 Code

While the graphical interface is fairly complete, it was not the main thrust of this work and as such is not well designed. Most of the interface is contained in one large source file and makes use of some global variables and classes. This was done to speed

the program development at the expense of modularization and encapsulation. It is only intended as an example of what one could do for the interface and as a quick means to easily control the feeder.

The application makes extensive use of Swing components [30] to create the user interface. As such, a relatively new (as of Jan, 2002) Java virtual machine is required to run the interface (\geq Java2, Version 1.2).

Two asynchronous threads are used in the interface. The first responds to user events and sends requests to the feeder. The second thread listens to the socket for replies and updates from the feeder. When an update is received, the second thread decodes the message, and requests that the event-handling thread update the GUI. This is accomplished by passing a runnable method that performs the GUI update to the method `invokeLater()` of the class `SwingUtilities` [31].

Messages which are passed between the user interface and the feeder control are encoded as plain ASCII text. While this makes the messages a bit larger in size, it alleviates the need to be concerned with the endian-ness of the machine that is hosting the interface or the feeder controller. Messages are created and decoded using the `stdlib` functions `sprintf` and `sscanf` with the appropriate format strings.

Lastly, the interface maintains a local copy of the state of the feeder system. This is required because the GUI can only be updated from the event-handling thread. The thread which receives updates from the feeder writes the necessary changes to the feeder state data structure and then requests the GUI be refreshed. A piece of code, which executes in the context of the event-handling loop, then reads the state data and updates the GUI.

4.1.3 Login/Logout Sequence

Lastly, the sequence of events which occur in the user interface when a user logs in or out will be traced.

After starting the program, one needs to enter a user name and password. Also, the name of the machine on which the feeder controller is running and the listening port also needs to be entered (if it is different from the default). Next the user presses the login button and a socket is opened to the remote machine. The user name and password

are retrieved from the GUI and placed into a message. Currently no encryption is used, the username/password travel in plaintext. If the username and password are valid (determined by the feeder), then the feeder replies as such. The interface then creates the listening thread to receive system updates and waits for further input from the user. The feeder controller will send an update on its current state, which will be received by the update thread. The update thread will then refresh the GUI to match the current state of the feeder.

If the user entered an incorrect username or password, a pop-up window relays the message to the user and the interface waits for additional user input (i.e. retype the username and password and try to log in again).

When the user is ready to logout, the logout button is pressed. This sends a logout message to the feeder controller. The feeder controller sends a reply which is received by the update thread. The update threads sees that the user has logged out, makes a final update of the GUI (sets it back to its default *not-logged-in* state), closes the socket, and terminates itself. The GUI does not terminate; it waits for the user to either exit with the file→exit menu item or to log in to the feeder again.

4.2 remote_user Object

Whenever a user logs into the feeder, a new `remote_user` object is created to handle interactions with the user. The `remote_user` object is first responsible for ensuring the user presents the proper credentials (in the form of a username and password) and then handles all network communications between the user and the system. The object is also responsible for registering the user with the `main_control` object so that the user receives system state updates. Finally, when the user logs out, the object makes sure the user de-registers with the main controller and the communications socket is properly terminated before destroying itself. Table 4-1 lists the public methods of the class.

The `remote_user` object uses two asynchronous threads to accomplish its tasks. The first thread, created by the `main()` function when a request arrives at the listening socket (see the next section for an explanation of `main()`), is used to receive requests from the user and act upon those requests. Requests are decoded by the

`service_request()` method and the appropriate method call is made to the `main_control` object to accomplish the request. If the request was not valid (e.g. the user does not have control, the requested state change is not possible, etc.), the `service_request()` method returns an error to the user.

Method Name	Description
<code>login()</code>	Checks the username and password; registers the user with the <code>main_control</code> object; creates a <code>handle_updates</code> thread.
<code>get_requests()</code>	Retrieves a request from the remote user (from the socket) and places it into an internal data location.
<code>service_request()</code>	Decode the request and act upon it.
<code>static handle_updates(remote_user *ru)</code>	Method executed as an asynchronous thread which receives updates from <code>main_control</code> and sends them along to the user.

Table 4-1: `remote_user` Public Interface

A second thread, created in the `login()` method after the user's credentials have been verified, is used to send system updates back to the user. The static method `handle_updates()` is executed in the thread. This method listens to a message queue, created when the `remote_user` object registered with the main controller, for system updates. When an update appears, the method creates an update message and sends it to the user. A message queue is used as a buffer between the remote users and `main_control` to protect `main_control` from network delays and problems. For example, it would not be good for the system to be delayed in its primary task of parts feeding because it was stalled trying to send an update to a user. `main_control` simply places the update into the queue, it is up to the `handle_updates` thread to actually get the message to the user.

One caveat of this method is during logout. The destructor must send a special message to the `handle_updates` thread so that it realizes the user is logging out and terminates itself.

4.2.1 Login

This section examines the case of a user logging into the system. Figure 4-6 shows an interaction diagram for this situation.

The login sequence begins when the user presses the login button. A login request is sent to the feeder and received by the `main()` function. `main()` creates a new `remote_user` object and then creates a thread to service the user. `main()` then returns to listening for another user login. The new thread first calls the `login()` method of the `remote_user` object. `login()` first checks the username/password and then creates a new thread to handle updates sent to the user from the feeder control. `login()` next sends a `LOGIN_SUCCESS` message back to the user and then calls the `register_me()` method of the main controller which enables the user to receive system updates. `login()` finally returns `TRUE` to the calling thread which waits for a user request to arrive.

After `main_control` registers the new user, it sends a state update message to the `user_update` thread (described in the next section) which sends a state update to all logged in users, including the one that just logged in. This message causes each user's GUI to update to reflect the new state of the system.

4.2.2 User Logout

This section traces the sequence of events which occur when a user decides to logout of the system. Some details, such as checking to be sure the user does not currently have control, are skipped in the following discussion. Figure 4-7 shows the interaction diagram of the events.

thread notices this (it checks the variable each time before listening for additional requests) and calls the `remote_user` destructor. The destructor first calls the `unregister_me()` method of the main controller which removes the user from the update list (it also causes an update state message to be sent to all users logged in). The destructor next sends a `TERMINATE` message to the user update thread which causes it to terminate. Next a `LOGOUT_SUCCESS` message is sent to the user. Finally, the socket is closed and the `remote_user` object is destroyed. The thread dedicated to the user then terminates. This concludes the logout process on the feeder controller.

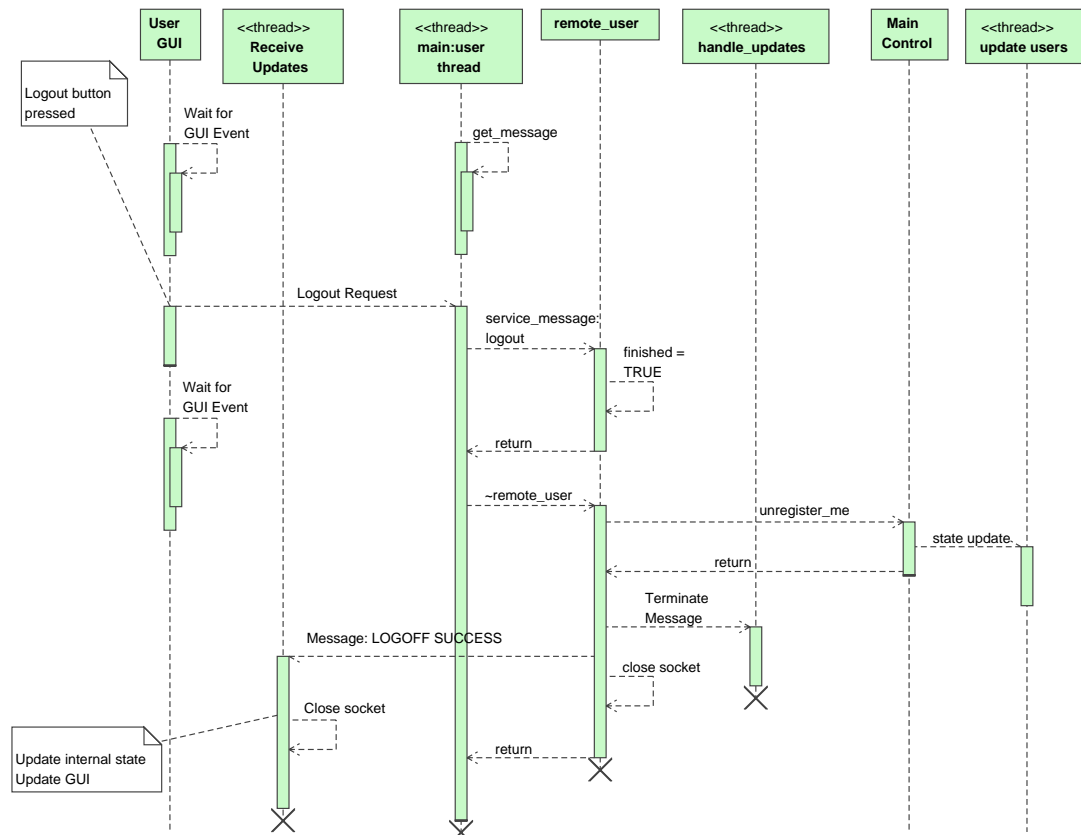


Figure 4-7: User Logout

On the user interface, the update thread receives the logout success message, resets the interface's state to the *not-logged-in* setting, closes the socket, and then terminates.

4.3 `main_control` Object

The `main_control` object is responsible for the overall control of the feeder. It communicates with the remote users, maintains the state of the system, ensures that transitions from state to state happen properly, creates the part-specific objects when needed, and makes sure the system is shutdown properly. The `main_control` object is concerned with the control of the feeder at a very general level. It simply knows if the feeder is `RUNNING`, `PAUSED`, or `STOPPED`, and what part is currently being fed (if the system is running). Finer details of control (how the feeder is accomplishing its task, what each component of the system is doing, etc.) are handled at lower levels of the control software.

The `main_control` object presents an interface (Table 4-2) to the rest of the control software that enables `remote_user` objects to affect changes on the feeder and allows mid-level components to report system data to the `main_control` object. Methods can be grouped into general categories: user housekeeping (e.g. registering, gaining and releasing control) feeder function changes (e.g. switch system state, set robot speed) data update (e.g. robot retrieved a part).

At start-up, the `main_control` object first creates and initializes the data structures which hold the current state and statistics of the system. Next, a thread is created, `update_users`, that becomes the central collection point for messages and updates to remote users. Any other task in the system may send updates to the users via a message queue maintained by this thread. By forcing all messages and updates to go through this thread, the data structures which contain the state of the system may be kept consistent. After this thread is created, four asynchronous threads are created (the asynchronous threads are discussed in detail in the next chapter). Three of the threads correspond directly to feeder hardware (robot, vision system, and conveyors) while the fourth thread is used by the auto-adjusting system. The threads are free running and watch the current state of the system (as contained in the `main_control` object) for information on what to do. There are three possible states for the threads (and, in general, the feeder): `RUNNING`, `PAUSED`, and `STOPPED`. At start-up, the threads are in the `STOPPED` state. Lastly, the `main_control` object waits for a user to log in and send a

command. Figure 4-8 shows an interaction diagram of the start-up and shutdown of the `main_control` object.

Method	Description
<p><i>User Housekeeping</i></p> <pre>register_me(thread_ID, *queue) unregister_me(thread_ID) control(thread_ID) request_control(thread_ID) relinquish_control(thread_ID)</pre>	<p>Methods to enable the users to interact with the <code>main_control</code> object; check control status, request and release control.</p>
<p><i>Feeder Control</i></p> <pre>set_hor_speed(double speed) get_hor_speed() set_inc_percentage(int percentage) get_inc_percentage() set_robot_speed(int speed) get_robot_speed() set_part_type(int part_type) change_system_operation(int new_state) set_auto_engage() get_system_state()</pre>	<p>These methods are used to affect changes on the feeder. They can be used to start and stop the system, to change the speeds of system components, and to change and report on the current system state.</p>
<p><i>Data Update</i></p> <pre>robot_update(int part_type, double time) robot_update(double time) refresh_system_state()</pre>	<p>The methods in this group are used by the mid-level components to inform the <code>main_control</code> object that an event has occurred, such as a part being retrieved.</p>

Table 4-2: `main_control` Public Interface

Finally, when the system is finished feeding, `main_control` ensures that objects are properly destroyed. The asynchronous threads must be terminated, any users which are still logged in must be notified that the system is shutting down and then logged out, and the `user_update` thread must be terminated. After this is accomplished, the `main_control` object itself may be destroyed. Proper shutdown of communication channels with the other hardware controllers in the system is accomplished through the destruction of the server-level objects described in Chapter 6.

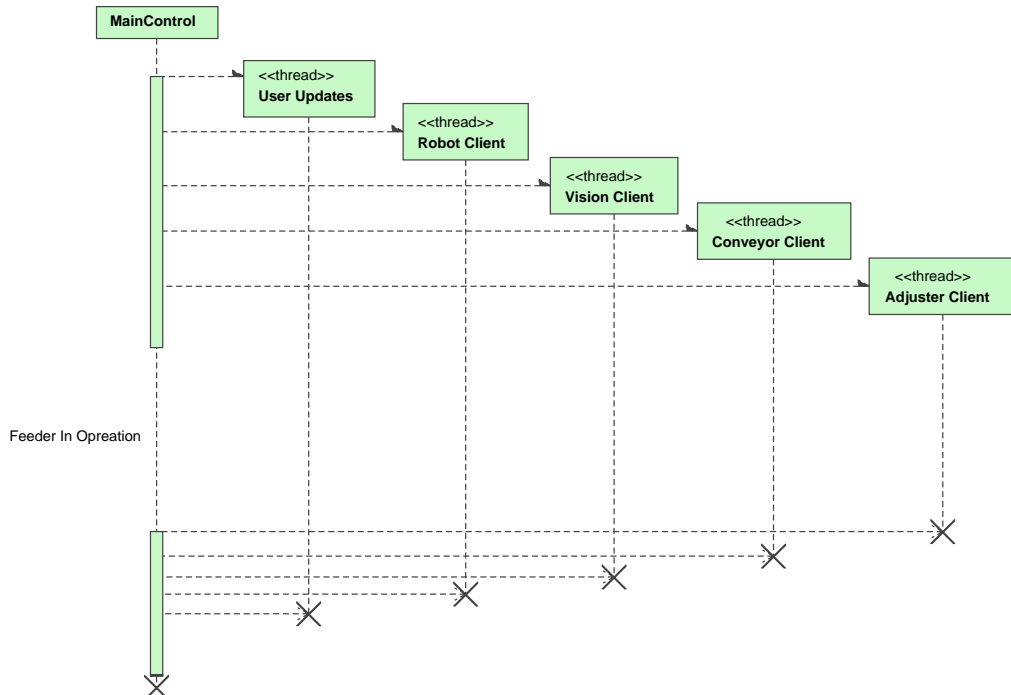


Figure 4-8: main_control Start-up and Shutdown Interaction Diagram

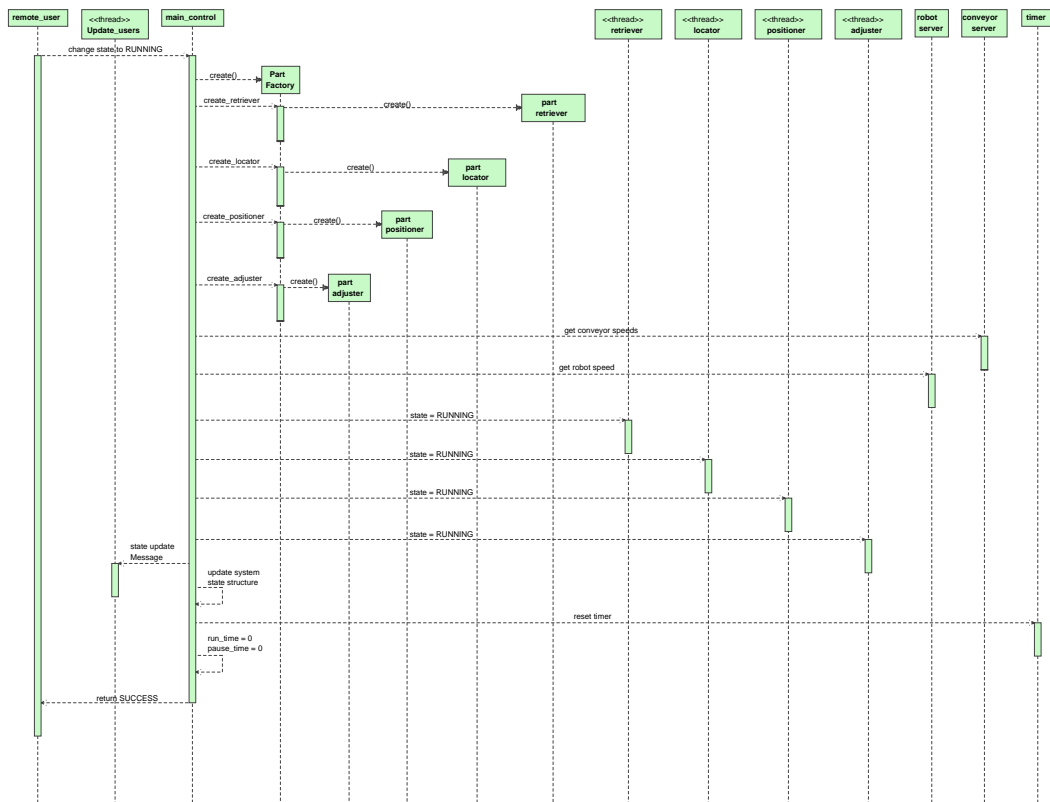


Figure 4-9: State Transition: STOPPED → RUNNING

The final major task of the `main_control` object is to transition the feeder from one state to another. There are three states in which the feeder can be: `STOPPED`, `RUNNING`, and `PAUSED`. While a number of state transitions are allowed, for brevity, only the transition from `STOPPED` to `RUNNING` will be examined here. (The other transitions are explained in the source code comments.) Figure 4-9 shows the interaction diagram of this state transition.

The state transition begins when a user requests the system to start feeding parts. The `remote_user` object calls the `change_system_operation()` method of the `main_control` object with an argument of `RUNNING`. After a few checks, the `main_control` object creates an abstract factory for the part to be fed. This is the only place in the code where explicit knowledge of the part type is required (excluding the part-specific objects or PSOs, described in the next chapter). Next `main_control` creates the four PSOs.

It is important to create the PSOs in the proper order. For example, image subtraction is often used to enable the vision system to locate parts more reliably. Image subtraction requires a *golden image* which shows only the backlit window, no parts may be present. The object responsible for locating parts using the vision system, during construction, moves the conveyors to clear the vision window of parts (through the conveyor server). This must happen before the object which drives the conveyors is created, because that object will set the operating parameters for feeding the part, which could be overwritten when the vision object moves the conveyors.

After the PSOs have been constructed, `main_control` retrieves the initial speed setting for the conveyors and robot, assuming that the PSOs have set default values for those parameters in their respective constructors. The four asynchronous threads are then told to begin operation, followed by notifying all logged in users of the system state change (via the `update_users` thread). Lastly, `main_control` populates its internal data structures with the new system state, resets its run timer, and returns `SUCCESS`.

4.4 System Start-up / Shutdown

The final topic to discuss in this chapter is the start-up and shutdown procedures. A simple `main()` function is used to instantiate all the system objects in a

controlled manner; to listen for user login or shutdown requests, and to shutdown the system in an orderly way. Figure 4-10 shows the pseudo-code of the main function. When the program is started, it first instantiates all the server objects in the system: those that interface with the various hardware controllers. This is done to ensure all the servers have properly started. If one of the hardware controllers had been left in an inconsistent state from a previous system crash, it is better to catch the problem at this point and handle it gracefully rather than have the system fail unexpectedly at some later time. The final object to be created in this step is the `main_control` object.

If the servers all start successfully, the program opens a socket to listen for login requests, prints an informative message to the console, and waits for a login or shutdown request. From this point, all interactions with the feeder must occur via the networked GUI. The only input allowed at the console is a shutdown request, accomplished by pressing *q* on the keyboard.

```

if(start_up_servers() != SUCCESS)
    exit -1;

soc = open_socket()
print System up and accepting connections;

while (continue)
{
    listen for login request or q keypress;

    if (login_request)
        *temp_user = new remote_user(soc);
        create_thread(temp_user);

    if (``q`` key press)
        continue = FALSE;
}

print System shutting down;

shutdown_servers();
exit 0;

```

Figure 4-10: `main()` Function Pseudo-Code

Eventually, when the user presses *q* on the keyboard (at the console), the system shuts down. First, the `main_control` object is destroyed, then the server objects are destroyed in an orderly manner. Finally, the program exits.

5 Mid-Level Components

Mid-level components in the software architecture can be separated into three groups: asynchronous threads, part-specific objects, and utility objects. The asynchronous threads are created by the `main_control` object at system start-up and only terminate when the system is shutdown. PSOs are created each time the system begins feeding a different part (transitions to the `RUNNING` state). They are destroyed when the feeder transitions to the `STOPPED` state. The utility objects are used by the other mid-level components and consist of an encoder synchronization object and a queue to hold parts which have been located but not yet retrieved. Both utility objects are created by `main()` at system start-up.

5.1 Asynchronous Threads

The asynchronous threads are four threads that run any time the feeder software is running. They perform functions specific to each sub-system of the feeder. One thread is responsible for controlling the conveyors, a second is responsible for controlling the vision system and finding parts, the third controls the robot and retrieves parts, and the last is responsible for running the auto-adjusting component of the feeder.

Each thread can be in one of three possible states (`RUNNING`, `PAUSED`, or `STOPPED`), which correspond with the state of the feeder. The threads run in infinite loops, each time through checking if the state of the system has changed. If the state is `RUNNING`, then each thread performs its specific task. For example, the vision thread finds parts in an image, the robot thread retrieves a part, etc. If in the `PAUSED` or `STOPPED` state, the thread simply sleeps for a second and then re-checks for a state change. Figure 5-1 shows this in pseudo-code:

While each thread is responsible for a particular function, the thread does not specifically implement methodology to accomplish that function; that is deferred to the PSOs. This allows the asynchronous threads to be completely generic and not tied to any specific part. When a part is selected to be fed by a user, the `main_control` object constructs the PSOs. The asynchronous threads always call abstract methods of the abstract base classes of the PSOs, which are bound to the concrete implementation after the concrete PSOs have been instantiated.

```

while(TRUE) {
    switch(state) {

        case RUNNING:
            perform required function;
            break;

        case STOPPED or PAUSED
            Sleep 1 second;
            break;

        case SYSTEM_EXIT;
            EndThread(*self);
            break;

    }
}

```

Figure 5-1: Asynchronous Threads Pseudo-Code

Each thread is actually a statically-defined method of the `main_control` class. When the `main_control` object creates the threads, it passes the respective function pointers to the thread creation API function. A pointer back to the main class is the single parameter to each method call. This allows the threads to access the base class's data structure, where the thread state information is retained.

In the `main_control` class, there are two copies of each thread's state. One that the `main_control` object wants the thread to be in (`thread_state_main`) and one that the thread is actually in (`thread_state_thread`). This allows a simple hand-shaking scheme to be employed to verify the threads are in the proper state. The `main_control` object first sets `thread_state_main` to the desired state and then watches when `thread_state_thread` also changes to that state. This is important when one needs to enforce an ordering to the threads as they change states.

5.2 Part-Specific Objects

The part-specific objects are the second component of the mid-level objects and are used to encapsulate all the specialization necessary for feeding each parts. This allows the rest of the control system to remain consistent no matter what part is currently being fed. A new set of PSOs are required for each new part or feeding scenario. (A combination of parts fed at once constitutes a new feeding scenario, even if all the parts have already been fed individually and have their own PSOs.) New PSOs

are required because the parameters and implementation of feeding multiple parts might be different from the single case. For example, a vastly different vision algorithm may be required to locate multiple dissimilar parts than the algorithm use to find each part alone.

A set of abstract base classes are defined that enforce a common interface in all PSOs. This ensures that all subsequent PSOs will function properly in concert with the rest of the feeder software. There are a total of five abstract base classes defined: one class factory which knows how to create the remaining classes which follows the abstract factory design pattern. The four remaining classes align with the four asynchronous threads (retriever, locator, positioner, auto-adjuster); each thread uses its corresponding PSO to accomplish its defined task.

Each PSO contains all the *know-how* required to perform its function. This includes all feeder parameters used to feed the parts: the speeds of the conveyors, the tool offsets used by the robot when retrieving the part, the exposure setting used by the camera when grabbing images of the parts, etc. Additionally, all algorithms specific to the part are also encapsulated. Examples include the method of part location (i.e. the vision algorithm). Does it use gray scale or binary processing? Does it apply feature comparison for recognition (e.g. area, perimeter)? Does it use a pattern matching technique? An additional example would be the auto-adjusting algorithm. What parameters are adjusted? What parameters are used as metrics when determining changes? Is any modeling applied to the adjusting controller?

Figure 5-2 shows the class diagram for the PSOs. As shown, the asynchronous threads only accesses the PSOs through the interfaces declared by the abstract classes. A concrete implementation of a part factory is instantiated when the user changes the system state from STOPPED to RUNNING. The specific factory is determined by the currently selected part. When the system changes from RUNNING to STOPPED, the factory (and its PSOs are destroyed).



Figure 5-2: Part Specific Object Class Diagram

5.2.1 Constructors / Destructors

The constructors of the PSOs are used to handle initialization for each feeding situation. For example, when feeding lenses, image subtraction is used. Therefore, a *golden image* is required (an image with no parts). This is accomplished in the constructor of the `lens_locator` object. It clears the horizontal conveyor of parts and then grabs the image. Initialization common to all similar PSOs can be placed in the constructor of the abstract base class (e.g. get a pointer to one of the server-level objects).

Complementary to initialization in the constructor is proper clean-up in the destructor. It is important to ensure that anything started up in the constructor is

properly stopped in the destructor. For example, if backlighting is needed in the locator, then the locator constructor turns on the backlight. It is important then to remember that the destructor must turn off the backlight or else it will be left on. If the next part does not need backlighting, then when the feeder is started for that part, the backlight will be in the incorrect state which may cause the locator to fail to find parts.

5.2.2 Common Functionality

While each PSO is specific to a particular part, there are some cases where a common functionality is required by all PSOs of the same type. This functionality may be implemented in the abstract base class and then inherited by each specific PSO. An example is the `too_far()` routine in the retriever PSO. This routine is used to ensure that the part the retriever is attempting to retrieve has not fallen from the end of the conveyor. All retrieve PSOs need this functionality, so the routine is written once and used by all. An added benefit is if/when a better routine is written for `too_far()`, it is immediately available for all PSOs to use.

A second example could be in the auto-adjuster PSO. If it is determined that a certain algorithm is always needed when computing a better set of feeder parameters, then it may be included in the auto-adjuster abstract base class and be available for all auto-adjusting PSOs to use.

5.2.3 New Part Introduction

A concern with the inclusion of PSOs into the software design is how much new code must be generated each time a new part or scenario is introduced into the system. One must create concrete implementations of the five abstract base classes. While this seems like it might be a large task, thus far in practice it has been straightforward. Each new part introduction consisted of approximately 500 — 1000 new lines of code. Most of that was in the composition of a new vision routine for locating the new part. Much of the standard code was able to be copied over from a previous instance of the PSOs. This is contrasted with the total code base of approximately 25,000 lines; 2% — 4% new code per new feeding scenario.

5.3 Utility Objects

Two utility objects are utilized at the mid-level. These objects provide necessary functionality to the other mid-level components. The first object, the conveyor encoder object, is responsible for synchronizing the encoder value of the horizontal conveyor encoder between the Galil controller and the Adept controller. The second object is an implementation of a FIFO queue used to hold parts which have been located but not yet retrieved. The locator PSO places parts into the queue and the retrieved PSO removes them from the queue for retrieval.

5.3.1 Conveyor Encoder Synchronization

As mentioned briefly in Section 2.3, the Galil conveyor controller uses a 32-bit signed number to keep count of the encoder on the horizontal conveyor. The Adept robot controller (also watching the encoder signal) uses a 24-bit signed number for the same purpose. To enable the robot to properly reach parts on the conveyor which have been located by the locator PSO, the encoder value when the image was grabbed must be known. Using the latch line, it is easy to record the value of the encoder when an image is acquired. However, this value is given in Galil's encoding: a 32-bit signed number. For this value to be valid for the robot, it must be converted to the Adept representation. While one could simply note the difference in the two values at system start and apply this offset, this approach fails as soon as one of the encoder counters roll-over. In practice, a roll-over occurs every 5 — 15 minutes on the Adept counter and every 1 — 3 days on the Galil counter (depending on the speed of the conveyor).

The conveyor encoder object is responsible for converting the value of the encoder from the Galil representation to the Adept representation. It performs this function by computing the difference between the previous value and the current value of the Galil encoder and then applying that difference to its representation of the Adept encoder value. It is important to always account for roll-over (in both the encoder counters) in the calculation.

At creation, the object initializes its internal representation of the Galil and Adept encoder values. It is crucial that the conveyor not be moving for this to occur properly. A check is performed and if the conveyors are moving, the constructor fails. Each time a

method of the object is called, a calculation is performed and the internal encoder representations are updated.

As a consequence of this algorithm, if no method of the object is called for a period of time longer than required for the Galil encoder counter (technically, the larger counter) to completely wrap around past its current value, then the values will be out of synchronization and the robot will no longer be able to retrieve parts. Wrapping of the smaller encoder counter between calls is handled by performing a modulus operation before applying the difference to the smaller counter.

Table 5-1 lists the methods of the conveyor encoder object. The final responsibility of the object is to reset the latch on the Galil controller. It does this each time `enc_conv_to_adept()` is called. This method is used to determine the value of the Adept encoder counter each time an image is grabbed. The latch may also be reset by passing a `TRUE` value in the second parameter to the `read_latch()` method.

Method	Description
<code>enc_conv_to_adept(int *adept, int *galil)</code>	Return the current values of the Adept and Galil conveyor encoders, reset the latch.
<code>report_current_adept(int *adept)</code>	Report the current value of the Adept encoder counter.
<code>report_last_adept(int *adept)</code>	Report the value of the Adept encoder counter at the time of the last latch.
<code>read_latch(int *latch, bool reset)</code>	Return the value of the latch (1: latch not tripped, 0: latch tripped), reset the latch if desired.
<code>instance()</code>	Static <code>instance()</code> method used in the singleton design pattern.

Table 5-1: Conveyor Encoder Public Interface

To ensure that only a single copy of the conveyor encoder object is present in the system, the Singleton design pattern is used. Because the system is a preemptive, multi-

threaded environment, the double-check guard adaptation of the singleton pattern is implemented.

5.3.2 Parts Queue

The parts queue is a basic FIFO queue which allows the locator PSO and the retriever PSO to operate independently. The locator finds parts and places their locations in the queue; the retriever removes locations and retrieves the respective parts. Neither PSO need be concerned with the current operation of the other. The retriever PSO is responsible for ensuring the part has not fallen from the end of the conveyor (using the `too_far()` method).

Internally, the queue is constructed as a singly-linked list. A node is composed of a `CPart` object and a `next` pointer. A condition variable is used to control access to the queue. Table 5-2 lists class methods.

Method	Description
<code>put(CPart *part)</code>	Put a <code>CPart</code> object onto the queue.
<code>put_front(CPart *part)</code>	Put a <code>CPart</code> object onto the front of the queue (the next part removed will be this part).
<code>CPart* get()</code>	Get a <code>CPart</code> object from the front of the queue.
<code>flush()</code>	Clear the queue.
<code>int how_many()</code>	Report the number of <code>CPart</code> objects on the queue.
<code>instance()</code>	Static <code>instance()</code> method used the singleton design pattern.

Table 5-2: Part Queue Public Interface

The `put_front()` method is used to get the attention of the retriever thread. Normally, when in the `RUNNING` state, the retriever simply waits for a part to appear on the queue and then proceeds to retrieve it. When the system transitions from the `RUNNING` state to the `STOPPED` state, a special `CPart` object is placed at the front of the queue which ensures the retriever thread will not block waiting for a part to appear on the queue.

Consider that if the locator thread is already stopped, a part will never be placed on the queue. Hence, the retrieve thread will never return from waiting for another part

and therefore never enter the `STOPPED` state. Two problems result. First, the retriever thread could have a stale pointer to its `PSO` when the next part to feed was selected and the system started. Second, because the `main_control` object monitors the state of each asynchronous thread, the system would never reach the `STOPPED` state.

Like the conveyor encoder object, this object also uses the double-check guard modified singleton design pattern for its creation.

6 Sever-Level Components

There are five major server-level components in the system. They are the robot motion server, the robot position server, the vision system server, the conveyor motion server, and the digital I/O server. Each server is created using the singleton design pattern with the double-check guard modification. This ensures only a single instance of each server is created and provides a simple means for other system components to access the servers. Server-level objects are created by the `main()` function at system start-up and are not destroyed until the system is shutdown.

Servers perform several important functions. First, they encapsulate each piece of hardware and present the rest of the system with a generic interface to that component. This allows the hardware to be replaced (at a later date) with a similar component without rewriting the entire code base; only that specific server must be rewritten. They also hide the communication details between the server and the physical hardware. For example, the robot server communicates with the Adept controller via Ethernet. If a different communications method was employed (e.g. a reflective memory network as with the original CWRU feeder), then only the server would be aware of this fact and require alteration. When combining these two functions, one could consider the servers a hybrid of the proxy design pattern [26c] and the adapter design pattern [26d]. Lastly, servers provide access control to the hardware. Unlike a re-entrant function, the physical hardware components can only do one thing at a time. Thus, in a multi-threaded environment, it is crucial to prevent two threads from simultaneously instructing the hardware to perform incompatible tasks. By providing this access control at the server-level (through the use of monitors), the rest of the system need not be concerned with causing physical problems by issuing conflicting instructions.

6.1 Robot Motion Server

The robot motion server is used to control the Adept Cobra600 robot. The robot is physically attached to the Adept AWE controller. The server communicates with the Adept controller via an Ethernet socket. The socket is opened and a connection setup in the server's constructor. If the server is unable to communicate with the Adept controller, the constructor fails. A special program (described in Chapter 7) is run on the

Adept controller to receive and execute commands from the robot server. Many server methods may be blocking or non-blocking depending on the parameters in the method call. Table 6-1 shows the public interface of the object. Methods may be grouped into several general categories (parameter lists have been omitted for clarity; see [19], [20] for complete interface documentation).

Method	Description
<p><i>Motion</i></p> <p>move() pick_and_place() pick_from_belt() approach() depart()</p>	<p>Methods that cause the robot to move to a location or perform a specific task. Most methods require position information to be given in their parameter lists.</p>
<p><i>Parameter Set/Get</i></p> <p>set_accel() report_accel() set_decel() report_decel() set_speed() report_speed() set_configuration()</p>	<p>Methods which are used to set or return operational parameters of the robot. set_configuration() is used to specify a lefty or righty configuration of robots whose kinematics allow multiple solutions to any given position.</p>
<p><i>Control</i></p> <p>enable_dry_run() disable_dry_run() attach() detach()</p>	<p>Methods to affect the operation of the robot and its controller.</p>
<p><i>Tool Offset</i></p> <p>set_tool_gripfix() set_tool_null() set_tool()</p>	<p>Methods which allow tool offsets to be defined and enabled on the robot controller.</p>
<p><i>Other</i></p> <p>RunCmd() check_pos() get_pos() instance()</p>	<p>Various other methods for interfacing with the robot. RunCmd() is a back-door to allow arbitrary command execution. Static instance() method used in the singleton design pattern.</p>

Table 6-1: Robot Motion Server Public Interface

6.2 Robot Position Server

The robot position server is used to return the current location of the robot. This functionality is provided outside of the robot motion server so that other system components may get the instantaneous robot location regardless of the current operation of the motion server (e.g. it may be in the middle of a blocked motion command). The server communicates with the Adept system via an Ethernet socket (different than the motion server). A program is run on the Adept controller to respond to position requests. An additional function of the position server is to return the value for the horizontal conveyor encoder as viewed by the Adept system. This is needed when constructing the conveyor encoder utility object. Table 6-2 shows the server's public interface.

Method	Description
<code>get_current_position(*loc)</code>	Return the current location of the robot in the provided location structure.
<code>get_current_belt_encoder()</code>	Return the current value of the conveyor encoder as viewed by the Adept controller.
<code>get_current_belt_xform(*belt)</code>	Return the coordinate transform used to define the location of the belt relative to the robot.
<code>instance()</code>	Static <code>instance()</code> method used in the singleton design pattern.

Table 6-2: Robot Position Server Public Interface

6.3 Vision System Server

The vision system server is used to control the Matrox vision system (Matrox Meteor II frame grabber on the PCI bus). It is used to grab images from the camera, to analyze those images, and to return the locations of parts which have been identified. Internally, it makes extensive use of the MIL in performing its functions. Most of the methods simply perform the proper sequence of MIL calls to accomplish the desired tasks.

The server is also responsible for performing the proper initialization sequence for the vision system. The Matrox system must be properly setup (in software) before it

may be used and these functions are executed in the constructor of the server. This ensures the system is ready to go when the constructor is finished.

Lastly, the vision server is used to perform the calibration between the robot and the vision system. During a setup procedure, the physical (geometric) relationship between the camera and robot is determined. After that calibration, the vision system finds and reports the locations of parts directly in the world coordinate frame of the robot. The locations, as reported by the vision system, may then be passed directly to the robot for part retrieval.

Table 6-3 lists the public interface of the vision server. Methods can be grouped according to general functionality. Parameters in method signatures are omitted for clarity (see [19], [20] for complete documentation of the vision server).

Method	Description
<p><i>Image Acquisition</i></p> <p>GrabImage() GrabContionuous() GrabHalt() DisplayImage() UnDisplayImage()</p>	<p>Methods to grab an image from the camera and display it in a window on the screen.</p>
<p><i>Image Analysis</i></p> <p>SubtractFromBackground() Threshold() FindBlobsByArea_Perim() FindBlobsByArea_Perim_Mom() FindBlobs()</p>	<p>Methods to analyze an image and locate parts in it.</p>
<p><i>System Settings</i></p> <p>SetExposure() GetExposure() GetImageBuffer() DeletImageBuffer() GetFrameRate() SetFrameRate()</p>	<p>Methods to set system parameters and allocate frame buffers.</p>
<p><i>Utility</i></p> <p>DrawCross() DrawRectAtAngle() CalcAndDisplayHistogram()</p>	<p>Methods to draw on the image shown on the screen which are useful for algorithm development.</p>

Continued next page ...

Method	Description
<p>Calibration</p> <p>CalculateCalibration() FindCalibrationBlob() SaveCalibration() LoadCalibration() AssociateCalibration() TransformWorldToImage() TransformImageToWorld() TransformAreaImageToWorld()</p>	<p>Methods related to calibrating the camera to the robot. Methods to save and enable calibrations. Methods to convert between image coordinates and robot (world) coordinates.</p>
<p>Other</p> <p>GetTimerResolution() StartTimer() ReadTimer() EnableOverlay() instance()</p>	<p>General utility methods. Methods to access the high-resolution hardware timer on the Meteor II frame grabber. Static instance() method used in the singleton design pattern.</p>

Table 6-3: Vision System Server Public Interface

6.4 Conveyor Motion Server

The conveyor motion server is used to interface with the Galil 1822 motion control card (located on the PCI bus). It provides methods to control the inclined and horizontal conveyors used for part presentation. The server was ported from the conveyor server used on the original CWRU feeder (Galil stand-alone controller, serial communications).

The server also provides the callback service to notify a thread when the horizontal conveyor has moved a specified distance. This is used by the mid-level vision thread to signal that it is time to grab another image and look for more parts. The callback is implemented through a message queue. The thread which wishes to receive notification provides the conveyor server with a pointer to a message queue and a distance. After this, messages are placed in the queue each time the conveyor moves the requested distance.

Lastly, the server provides an interface to the digital I/O points physically located on the Galil controller. Methods are provided to directly set or read an I/O point. A callback service is provided to send notifications when a digital input changes state. The

callback is implemented through a message queue in a similar manner to the callback service of the distance notification.

Internally, the notification functionality is implemented through interrupts. A simple program running directly on the Galil controller causes a hardware interrupt on the PC if a digital input changes state or when the conveyor moves the specified distance. This interrupt is caught by the Galil library software (provided with the controller) and is converted to a Windows message. An asynchronous thread, started when the conveyor motion server is created, listens for these messages, converts them to message objects, and passes them along to other threads which have properly registered. If no threads have registered to receive the messages, they are discarded. Only one thread may register to receive distance notifications (usually the mid-level vision sub-system thread) and one thread may register to receive input notifications (usually a thread in the I/O server).

In contrast to the robot motion and position servers (where local Adept programs reside permanently on the Adept controller), the program which runs locally on the Galil controller is downloaded each time the conveyor server object is created. The Galil hardware has primitive provision for storing programs locally (an area of flash memory), so to enable the code to be properly backed up and tracked, it is stored on the PC and downloaded when needed. A special method is used to download a program to the Galil controller. It allows for comments and formatting to be inserted in the program file to make it more human readable. By default, Galil allows very little flexibility in the format of program files. Details of relaxations in the Galil format are documented in the source.

While the callback function requires the use of a locally executing program on the Galil controller, most of the functionality of this server is implemented by simply sending commands directly to the Galil hardware.

Table 6-4 lists methods used by the conveyor motion server. They are categorized by general functionality. Parameters are omitted for clarity (see [19], [20] for completed details about method calls).

Method	Description
<p>Conveyor Motion</p> <pre> move() jog() jog_start() start() sync_move() shake() stop() all_stop() </pre>	<p>Methods to cause motion of the conveyors. Motion may be intermittent or continuous depending on the method used.</p>
<p>System Settings</p> <pre> accel() decel() speed() set_params() report_accel() report_decel() report_speed() report_position() report_slew_speed() report_all() report_var() </pre>	<p>Methods to set and read system parameters, speed, and accelerations of the conveyors.</p>
<p>Digital I/O</p> <pre> set_io() get_io() </pre>	<p>Methods used to set and report digital I/O points.</p>
<p>Callback</p> <pre> download() register_distance_thread() register_io_thread() unregister_distance_thread() unregister_io_thread() </pre>	<p>Methods related to callback notifications.</p>
<p>Other</p> <pre> instance() </pre>	<p>Static <code>instance()</code> method used in the singleton design pattern.</p>

Table 6-4: Conveyor Motion Server Public Interface

6.5 Digital I/O Server

The I/O server is used to provide a consistent means of access to all digital I/O points in the system. It also presents a uniform numbering scheme of the I/O points to the other system components. Physically, I/O points are located on the Galil motion controller and the Adept AWC controller. Access to the I/O points on the Galil controller

are provided through the conveyor server while the object communicates directly (via two Ethernet sockets) with the Adept controller. Internally, the server maintains a mapping of the current status of output points.

The I/O server provides two ways of accessing I/O points. The first is through traditional `set_io(num)` and `get_io(num)` routines. Each I/O may be toggled or queried independently. When performing a `get_io` operation, the server examines the I/O number and determines, physically, where the I/O point is located. It then reads the current status of the point via the conveyor server or the Adept controller and returns the result. When performing a `set_io` operation, the server first consults its local output map to determine if the output is already in the requested state. If so, the method simply returns. If the output is not in the proper state, then the server makes a request to either the conveyor server or the Adept controller to perform the operation.

The second method of accessing digital inputs is through a callback scheme. Any interested objects may register to be notified when a digital input changes state. When that input later changes, a message is sent to the thread (via a message queue) that indicates the input number and whether it was triggered by a rising or falling edge. Any number of objects may register to receive notification on a single input point and a single thread may register for as many inputs as it requires.

6.5.1 Public Interface

The class presents a simple public interface to the rest of the system for use. Table 6-5 shows the interface.

The `thread_ID` of the object which is registering (or unregistering) to receive input change notification is used to uniquely identify itself. If multiple objects are registered to receive notification for a given input and one object wishes to no longer receive updates, it calls `unregister_io_notify`. The server must be able to determine which object (from among those registered) to remove from the list.

Method	Description
<code>get_io(num)</code>	Method which reports on the current state of the specified I/O point.
<code>set_io(num)</code>	Method which sets the state of the specified I/O point (+num → enable, -num → disable).
<code>register_io_notify(*queue, thread_ID, num)</code>	Method which allows an object to register to receive notification of input state changes.
<code>unregister_io_notify(thread_ID, num)</code>	Method which enables a registered object to unregister itself.
<code>instance()</code>	Static <code>instance()</code> method used in the singleton design pattern.

Table 6-5: I/O Server Public Interface

6.5.2 Implementation of the Callback Functionality

The callback functionality of the I/O server is implemented using two asynchronous threads and a helper class, `io_interest`. The first thread is used to receive notification from the Adept controller via an Ethernet socket. A special program is run on the Adept controller that watches for input state changes and sends a notification when a change occurs. The thread translates the notification into a message which is subsequently placed in a queue maintained by the second thread.

The second thread uses a message queue to collect all input state change messages. Adept messages come from the first thread, input change notification from the Galil controller comes from the conveyor server. When the I/O server is created, it hands a pointer to the second thread's message queue to the conveyor server in a call to the server's `register_io_thread` method. Whenever an input changes on the Galil controller, a message is placed in the queue.

The second thread simply watches the queue for messages. When one appears, it converts the input number from the native Galil or Adept numbering scheme to the object's uniform numbering scheme and then calls the `notify` method of the `io_interest` object.

The `io_interest` object is used to maintain a list of all objects which have registered to receive notification. Internally, the `io_interest` object uses an array of

pointers to nodes. Each node denotes a registered object and holds the thread's ID of the object, a pointer to an object-supplied message queue, and a pointer to the next node. Each bin of the array corresponds with one input point in the system. As objects register to receive input notification, singly-linked lists of nodes are constructed, extending from each array bin. To notify interested objects, the `io_interest` object simply steps through the linked list of the corresponding input number and places a notification message in each registered thread's message queue. A `NULL` pointer is used to denote the end of a list. At creation, all the array bins are initialized to `NULL` pointers. Figure 6-1 illustrates the data structure. For example, in this case, two objects have registered to receive notifications for input number 1, three objects receive updates for input 4, while no objects are interested in input number 3, etc.

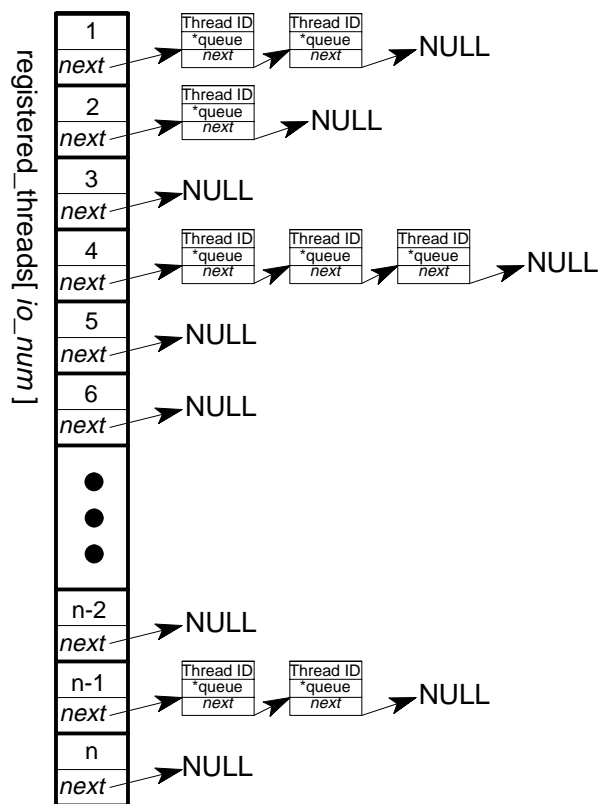


Figure 6-1: `io_interest` Registered Thread Data Structure

Figure 6-2 shows the sequence of events which occur when the I/O server object is created. The constructor first grabs an instance of the conveyor server. Next, a socket

connection is made to the Adept controller. The Adept controller spawns a task to handle digital input change notifications. The task initially waits for one second and then connects to a socket which the I/O server opens. The one-second pause is to allow the I/O server time to create the listening socket.

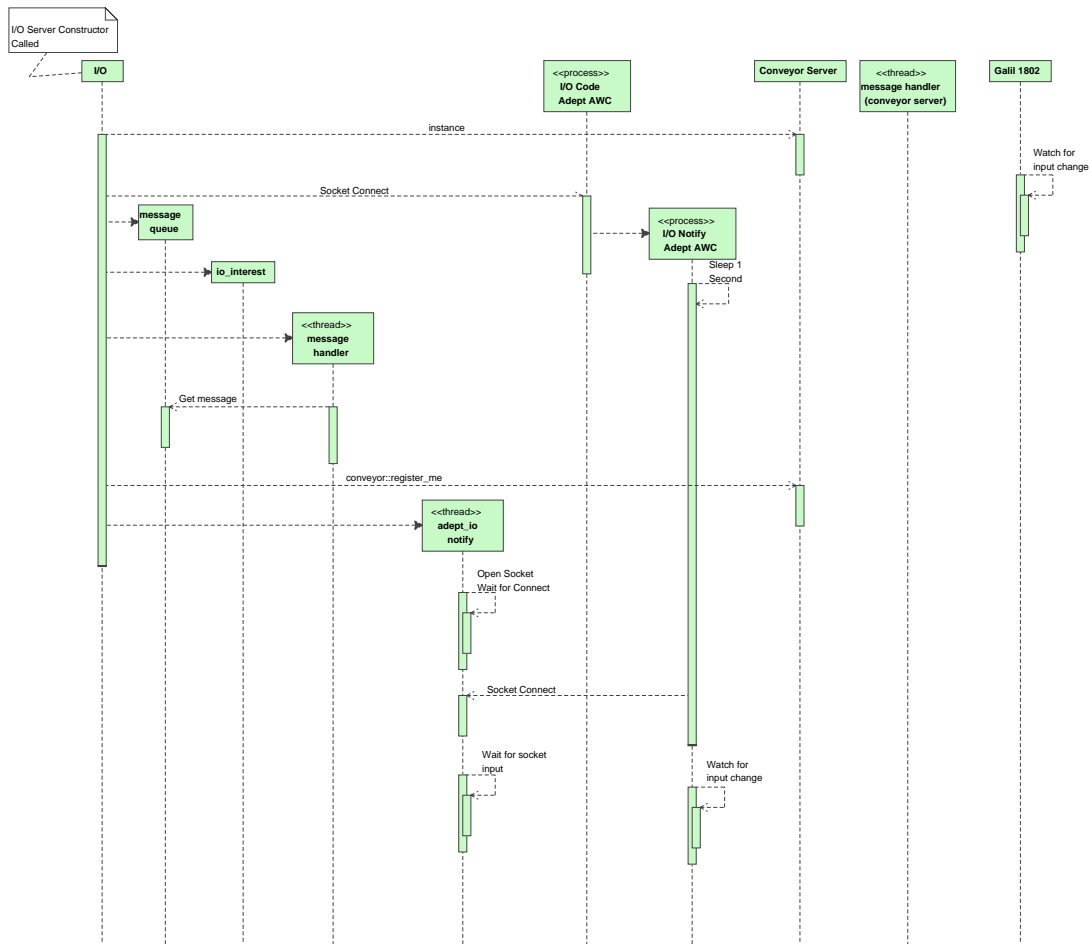


Figure 6-2: I/O Server Start-Up Interaction Diagram

Next, the constructor creates a message queue, an `io_interest` object, and creates a thread to handle incoming input change messages. That thread along with a pointer to the message queue is registered to receive input notification from the conveyor server. Next, a second thread is constructed to receive input notification from the Adept controller. It creates a socket and then listens for the Adept system to connect,

which it does when the one second pause is over. The system is now ready to receive and process input change notification.

Figure 6-3 shows the interaction diagram of events which occur when an input changes on the Galil and Adept controllers. The case with the Galil controller is shown and discussed first. The sequence is initiated when one of the inputs on the Galil controller changes state. This causes the program executing on the controller to generate an interrupt which is caught by the Galil library and converted to a Windows message. The conveyor server message handler thread catches the message, notes that it is in regard to an input change, and puts a message into the queue provided by the I/O server. The Galil controller continues to watch for further changes. The I/O server's message handler thread retrieves the message, converts it to the uniform numbering scheme, and calls the `notify()` method of the `io_interest` object. The `io_interest` object then checks the input number and places a notification message into the message queue of each thread that is registered to receive an update pertaining to that input. The message handling thread then waits for another message.

The second case examined is the Adept controller. The sequence begins by an input changing state on the Adept controller. This is noticed by the I/O notify process running on the controller, which generates a change notification and sends it to the `adept_io_notify` thread on the PC via Ethernet. The I/O notify process on the Adept controller then continues to wait for other inputs to change. The `adept_io_notify` thread receives the notification, constructs a message and places it in the I/O server's message queue. The I/O server's message handler thread retrieves the message, converts the input number to the uniform numbering scheme, and then calls the `notify()` method of `io_interest`. `io_interest` scans its array of registered objects and sends input change notifications to registered objects. The message handler thread then waits for another change message.

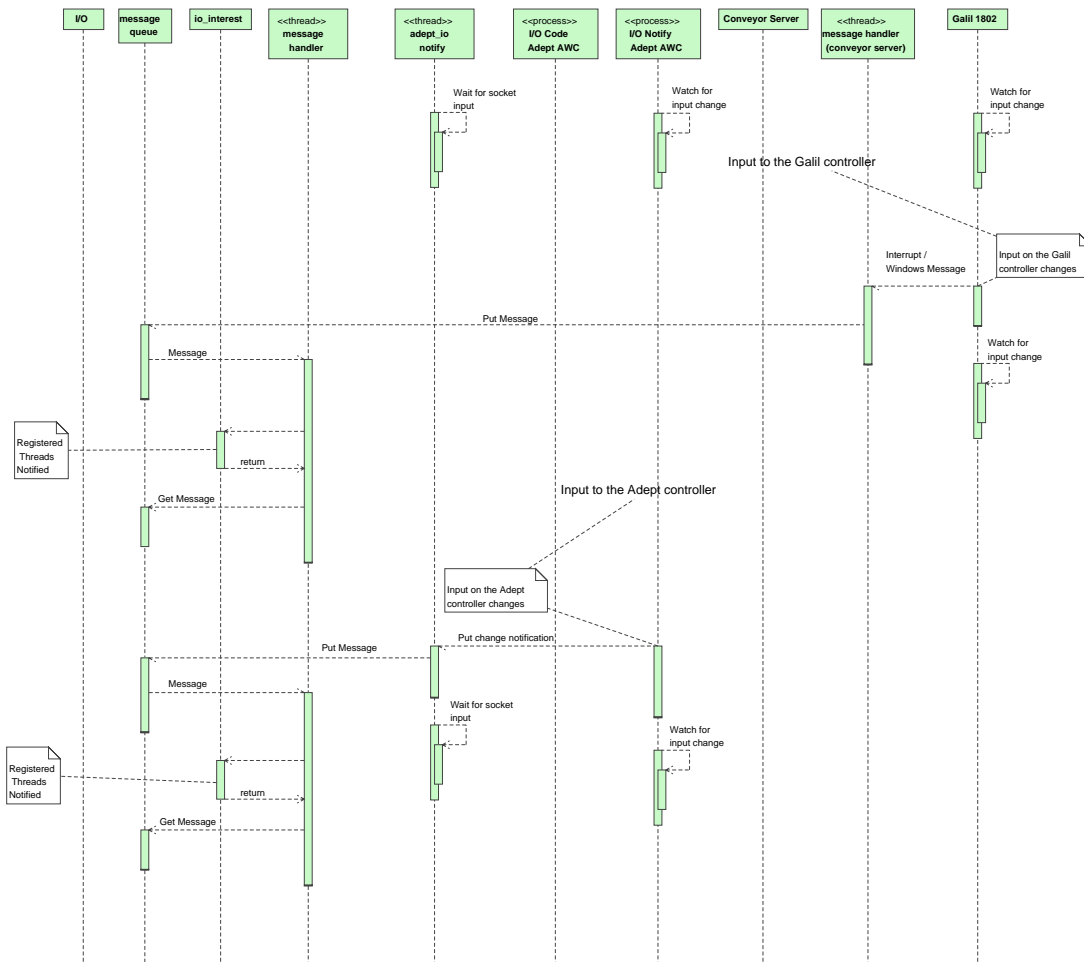


Figure 6-3: Input Notification Interaction Diagram

Multiple asynchronous threads are used to minimize latency in input changes to thread notification. By designating one thread to listen exclusively to the Adept system and a second thread to listen to both the first thread and the conveyor server, input changes are propagated rapidly through the server and to registered objects. If only a single thread were used, it would have to continually switch between checking the socket for a notification from the Adept controller and the message queue for a notification from the conveyor server. Two threads allow both notification sources to be watched simultaneously. Passing the Adept notification through the second thread's message queue provides a single point where all notifications pass, so that internal data structures may be kept up to date.

7 Hardware-Level Components

At the lowest level of control is the hardware. Each hardware component (conveyors, camera, or robot) has associated with it a dedicated controller or control board. The controller or control board oversees the operation of the hardware at the servo-level. Using dedicated, hardware-specific controllers, tight servo-level timing requirements are removed from the overall control scheme and placed into an environment where they are guaranteed to run at the required rate. This ensures the stability of the system; for example, the robot might end up waiting on the next motion command request due to a delay in the high-level controller, but it will not lose control of the robot because the servo loop failed to update in time.

Pushing the lowest level of control to vendor-supplied controllers also leverages the specific expertise of each vendor in their respective specialized areas. For example, one can be confident that Adept can adequately control its own robot; or that the Matrox frame grabber is capable of capturing images from the camera.

Each controller is only aware of its specific hardware component and only interacts with its respective server. It responds to simple commands, such as “robot: move to this position”, “conveyor: servo at 10 mm/sec”, or “GALIL: generate an interrupt when an input changes state” and has no knowledge of overall system goals.

While most of the control system was programmed using C++, the individual hardware controllers were programmed using their vendor-supplied languages: Adept uses V+, Galil uses a simple two-letter command set, and Matrox uses C function calls to the MIL. In addition, the location of the controller also affected the way in which the controller was programmed; programs were either run directly on the hardware’s CPU or individual commands were generated by the server-level objects and sent to the controller one at the time.

The following three sections examine each hardware controller and how it was programmed in detail.

7.1 Galil 1822 Motion Controller

The Galil controller is located on the PCI bus and is programmed using Galil's built-in, two-letter programming language. Programs may be run directly on the controller or single commands may be executed as they are sent to the controller. For programs executing locally, there are 8 tasks available for use.

The conveyor server uses both locally running programs and single commands to accomplish its task. Most of the functionality is implemented using single commands. For example, when a component calls `conveyor::move(hor, 100)`, indicating the horizontal conveyor should move 100 mm, the server constructs the correct command in Galil's language ("IP 100") and sends it to the controller (IP stands for "increment position"). The Galil library function `DMCWriteData()` is used to send the command to the card. The Galil library function `DMCReadData()` is used to check that the command executed correctly.

While this technique works well for most commands, a locally executing program is necessary for monitoring the state of inputs and for generating notification from horizontal conveyor motions. Two separate programs, downloaded to the controller at start-up, are used for this purpose. The programs run in different tasks simultaneously. The first program simply runs in a tight loop watching the input. It maintains the state of the input in a local data structure and generates an interrupt whenever an input changes. The second program is used to monitor the location of the horizontal conveyor and generate interrupts when it has moved the specified distance. If a thread has not registered to receive conveyor motion updates, then this program does not execute; it only runs when someone is registered to receive updates.

Lastly, the horizontal conveyor position latch is implemented using a built-in latch function. By executing the correct sequence of commands ("CN, , -1" followed by "AL X") the latch is enabled and armed. Then when the latch is triggered, the encoder value is automatically recorded in a predefined variable location ("_RLS"). One may then read the value from that location at a later time. After reading the value, the latch may be reset by sending another "AL X" command.

Porting the conveyor server from the original code (under VxWorks with a serial connection and stand-alone controller) to this version (under Windows NT with PCI bus communication) was only a matter of replacing the `send()` and `receive()` methods of the server object. Because both controllers (the stand-alone and the PC card) used the same two-letter language for operations, no other parts of the server had to change. Accomplishing the port in less than a week's time sped overall system development considerably.

7.2 Matrox Meteor II Frame Grabber

The Matrox frame grabber also resides on the PCI bus and is accessed using functions of the MIL, which essentially becomes the programming language for the card. In contrast to the Galil card, it is not possible to run programs locally on the Matrox card. It is a frame grabber with some specialized hardware tailored for image operations (look up tables, low pass filter, adjustable gain setting, etc.). Thus, the vision system server consisted entirely of methods that called the proper sequence of MIL functions to accomplish the desired task.

One beneficial feature of the Matrox line of vision card is the interchangeability between different models. If one finds the low end Meteor II card is not fast enough to perform the desired functions, it can simply be replaced with a more capable card, the code re-compiled and operations continued. A single flag to the MIL library at compile time indicates what hardware is being used. By changing the flag, internally, the MIL can exploit whatever resources that particular piece of hardware has. For example, some Matrox frame grabbers have on-board DSP's to speed vision processing computations.

7.3 Adept AWC Controller

The Adept AWC controller is, by far, the most advanced hardware controller in the system. It is a full fledged, stand-alone computer that is often used to control entire assembly cells. The controller has its own full-featured OS, local storage (32 Mb flash drive), and complete TCP/IP stack. The controller mounts (via NFS) a hard drive from the PC (running Sun's Solstice PC NFS server). OS storage is on the flash drive, user program storage is on the NFS drive. This makes it easy to backup the Adept-specific

software during regular PC backups. The V+ operating system is multi-tasking and allows up to 27 simultaneous tasks to operate concurrently. A prioritized, equal-sized time slice scheme is used for program control. There are 16 time slices of 1 ms each in a cycle. At each slice, the highest priority task runs. Tasks can change their priorities for each time slice and can be set to only run in specified slices. The system is not preemptive due to the hard real-time nature of robot and machine control (although it is possible to schedule user task such that motion control tasks are affected).

The controller is programmed in Adept's V+ language. It is a Basic-like, interpreted language that, while lacking many of the features of a modern object-oriented language, is well suited to machine control. Unlike the Galil and Matrox controllers, all interactions with the Adept controller are through locally executing programs. Each service which the controller performs (robot control, robot position reporting, and I/O control) is via a separate program running on the controller. No single-command execution is performed.

7.3.1 V+ Sockets

Since the controller communicates with the PC via Ethernet, sockets are used extensively in the V+ code. While basic socket functionality is provided [32], they operate through a slightly different interface than the standard Berkeley-style sockets. When creating a listening socket under V+, one calls `ATTACH` followed by `FOPEN` to create and open the socket. After setup, `READ` is used for all subsequent calls, including waiting for a connection and reading data after the connection has been established. The return value of `READ` is different depending on whether the return is due to a connection (on an unconnected socket), due to more data arriving on an established socket, due to a close request, or due to an error.

To open a socket, one calls `ATTACH` followed by `FOPEN`. If `FOPEN` returns successfully, the socket is ready to pass data. Subsequent calls to `WRITE` will put data on the socket. This difference in socket semantics makes writing server style programs straightforward in V+, as will be seen in the following sections.

7.3.2 Overall Software Setup

The software which runs on the Adept system is designed to automatically load and begin execution when the user powers up the controller. This is accomplished via a special boot parameter in the controller's firmware [33]. Upon booting, the controller may be instructed to load a single program from a specific drive directory (flash drive, NFS is not yet mounted) and execute its contents. A program, "AUTO", was created that performs the basic tasks of powering up and calibrating the robot, mounting the NFS drive, changing to the proper NFS directory, and loading and executing the main programs used with the feeder. Due to safety issues, the user must press the *power enable* button to enable robot power. This prevents the robot from being started remotely with persons in the vicinity being unaware of the potential danger.

The software used on the controller is composed of several V+ programs. During operation, four programs run concurrently. At start-up, a single program `a.server` is executed (by AUTO) which prepares the controller for feeding. First, an initialization file is executed which loads system constants into memory for later program use.[§] Next, two Adept-supplied programs are used to load a stored configuration file for coordinating robot and conveyor motion. Lastly, three server programs are started: the robot motion server program, the robot position server program, and the I/O server program.

7.3.3 Robot Motion Server Program

The motion server program first opens a socket and then enters an infinite loop calling `READ` on the socket. The first call to `READ` blocks until a connection is made. A connection is acknowledged by `READ` returning with a value of 100. Once a connection is established subsequent calls to `READ` will return data as it is sent to the port. Each time data arrives, the program calls the subroutine `decode_and_do` which performs the request and returns with a success or error code. The return code is written to the socket and `READ` is called again. When `read` returns with a value of 101, it is an indication that the remote socket has been closed. The program then resets the socket (with the `FCMND`

[§]V+, by default, has a flat memory structure. All programs have access to all variables. While it is possible to localize variables, the `init_constants` function simply initializes a series of variables in memory that can later be referenced by other programs. Though this is convenient, care must be taken to ensure an assumed constant variable is not overwritten inadvertently.

command) and calls READ again waiting for another connection. If an error occurs (e.g. the PC-side program crashed and the socket was not closed properly), READ returns with an unknown value. The program then closes the socket, creates a new socket and calls READ on the new socket. This makes the program resilient to crashes of the PC-side software. Figure 7-1 shows the pseudo-code for the robot motion server program.

```

Open the socket
while(TRUE) {
  val = Read(socket);
  switch(val) {
    case 100:
      print Connection
      break;

    case 101:
      print Socket Disconnected. Resetting ...
      reset socket
      break

    case 1:
      call decode_and_do(socket_data)
      write(return_val);
      break;

    case ANY:
      print Socket Crash!
      close(socket)
      open(socket)
      break;
  }
}
end;

```

Figure 7-1: V+ Side Robot Motion Server Pseudo-code

7.3.3.1 *decode_and_do*

The *decode_and_do* subroutine is used to decode the data retrieved from the socket and perform the required action. First, the subroutine decodes the initial part of the packet of data to determine the type of request. Next, a switch statement is used to jump to the appropriate section of code. The rest of the data is decoded (now that the format is known) and the robot is instructed to perform the requested action. Boiler-plate V+ commands are used to perform the action; the packet only contains the data

(positions, parameter values, etc.) that is necessary to make the generic code specific. Finally, any return values are noted in the `return_val` variable and the subroutine exits. Figure 7-2 illustrates with pseudo-code.

```

request_type = decode(first part of data)
switch(request_type) {
case request_1:
    specific data = decode(rest of data)
    perform desired action
    return_val = results
    return
    break

case request_2:
    specific data = decode(rest of data)
    perform desired action
    return_val = results
    return
    break

... and so on ...

case ANY:
    print Unknown request
    return_val = UNKNOWN_REQUEST
    return
    break
}

```

Figure 7-2: `decode_and_do` Pseudo-Code

The final program used in conjunction with the robot motion server is a automatic recovery program. Under some circumstances, for a variety of reasons, the robot may physically run into something. This usually causes the robot servo amplifier to time-out, which is a fatal error. Using a special construct under V+, one can define a recovery program that executes a series of commands if this occurs. A special program was written, which under some circumstances, can recover from such an error automatically. There are other errors from which recovery is not possible. If the recovery program fails, then the system stops with a fatal error. If this happens, the entire feeder must be shutdown and restarted. In practice, a complete restart is rarely necessary.

7.3.4 Robot Position Server Program

The robot position server is basically a simplified version of the robot motion server. The program first opens a socket and then enters an infinite loop. `READ` is called waiting for a connection. After a connection is made, each call to `READ` returns with a request, which is serviced in the program. The requested data (either the robot position or the conveyor encoder value) is written to the socket and `READ` is called again. If `read` returns with a value of 101, the remote socket closed. The local socket is reset and `READ` is called again waiting for the next connection. An error causes the program to close the socket, open a new socket, and then call `READ` to wait for a connection.

7.3.5 I/O Server Program

The I/O server program is used to set output points and read input points on the Adept controller. The program is very similar to the previous two programs. The program opens a socket and then enters an infinite loop. `READ` is called and the program waits for a connection. When a connection is made, the program first executes the `io_notify` program and then calls `READ` to wait for requests. When a request arrives, the program decodes it and performs the desired action (toggle an output or read an input) and then returns data if required (an input state was read). `READ` is then called and the program waits for another request. If the remote socket is closed, then in addition to resetting the local socket, the `io_notify` program must be killed. If the remote socket is closed poorly (the PC-side program crashes), then the socket must be closed and opened and the `io_notify` program must be killed.

7.3.5.1 *io_notify*

The `io_notify` program is used to monitor the digital input and send messages to the PC if an input point changes. The program first sleeps for a second (as explained in the previous chapter) and then attempts to connect to the socket opened by the `adept_io_notify` thread on the PC (started by the I/O server). Once connected, the current state of the input is read (and stored in an array) and the program enters a loop. Each time through the loop, the current input is read and compared with the previous values. If any values have changed, a message is sent to the PC indicating the change. If multiple inputs have changed, multiple messages are sent, one for each input point.

The program then calls `WAIT` which causes it to suspend until the start of the next major cycle.[§] This prevents the polling nature of the program from blocking other programs that are ready to run. The program will become runnable again when the next major cycle starts (as much as 16 ms). Each time through the loop, the program checks a variable to see if it should exit. This variable is set by the I/O server program when the socket has closed. If the program is to exit, it closes the socket and then ends.

[§]A major cycle is defined as one complete traversal of the 16 time slices of the OS's scheduling algorithm.

8 Results, Future Work, Conclusions

The final chapter concludes the thesis by first presenting results of using the software with the new feeder. Next, suggestions for future work are proposed. Much work remains in the overall problem of generic control of vision-based flexible parts feeding systems. While this work is a solid start, there are certainly many opportunities for further research. Lastly, some conclusions are drawn.

8.1 Results

8.1.1 Feeding Scenarios

To date, two different parts have been fed in the system. The first part was a round flashlight lens approximately 2 inches in diameter. A black band was painted around the perimeter of each lens to make it visible to the vision system. The second part was a flat square approximately 2 inches on a side. One corner of the square was removed to create a rotational asymmetry. Both parts were handled by a simple suction cup gripper. Figure 8-1 shows the two parts with a scale for size reference.

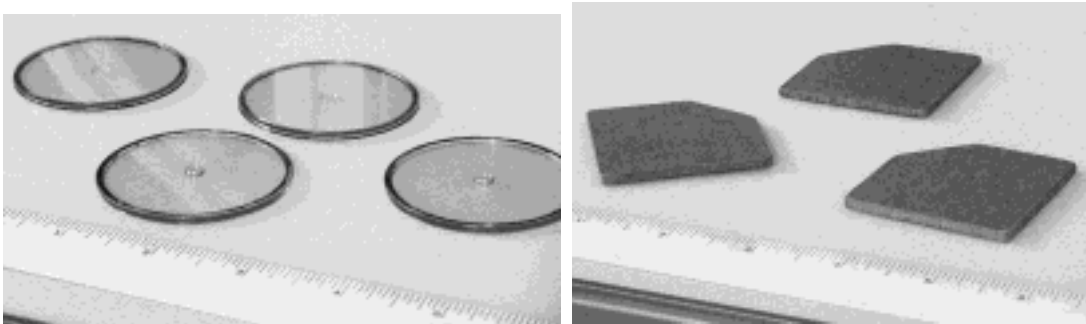


Figure 8-1: Test Parts: Lenses and Squares

Although there are only two different parts, three different feeding scenarios were tested: each part individually and a third scenario of both parts at the same time. Each scenario utilized a new set of PSOs tailored specifically for that situation. No other code was altered between scenarios.

Each set of PSOs was composed of approximately 1000 lines of code. However, much of that is due to documentation being included in the source files. When one looks at the amount of functional code for each set of PSOs, the number of lines is on the order

of 500 per PSO set. Reiterating, this represents about 2% new code vs. the total code base (~25k lines).

The system has also successfully switched between the three feeding scenarios without shutdown. After start-up, the feeder would be instructed to feed one of the parts. Then after a while, it was stopped, a new part selected, and then restarted. The system destroyed the initial set of PSOs on transition to the STOPPED state and then created a new set of PSOs for the next scenario when switched back to the RUNNING state. It was not necessary to shutdown the controller for this change-over to occur.

8.1.2 Throughput

Throughput testing was performed using only one feeding scenario; feeding the lenses alone. Other scenarios could not be tested due to an insufficient number of squares to properly fill the system. The squares were created as an example of an asymmetric part (to contrast the rotationally symmetric lens) and so a limited number were made in-house (~25 pieces). Several tests were run while throughput data was noted. Tests generally lasted from 6 to 10 hours duration. Overall throughputs in excess of 60 parts per minute were noted. This is in contrast to the original CWRU feeder which yielded a throughput of approximately 10 parts per minute when feeding the same part. (For reference, the original system fed smaller parts at over 30 parts per minute, however the throughput of this larger part was much lower.) This increase in throughput can be attributed to many factors including the parallel operation of the system, a faster robot, and a wider horizontal presentation conveyor.

8.1.3 Auto-Adjuster PSO

The last result was an attempt at implementing an auto adjusting PSO for the scenario of feeding lenses alone. This was also meant to be a general test of the functionality of the auto adjuster (beyond actually trying to create a good auto adjusting algorithm).

The initial try at an adjuster consisted of controlling the speed of the horizontal conveyor depending on the number of parts in the queue. The premise was that if there are a lot of parts in the queue, then the conveyor should be slowed so the robot has sufficient time to grab most of them. In contrast, if there are few parts in the queue,

then the conveyor should be run at a higher speed so that more parts can be quickly removed from the hopper and made available for retrieval. Unfortunately, this scheme proved to be ineffective. It ignored the fact that while many parts in the queue coupled with a very slow conveyor hurt throughput, it still satisfied the adjusting algorithm.

Secondly, the non-collocation of the feedback source (parts located by the vision system upstream of the robot) and the actual item wanting to be controlled (system throughput) caused some less-than-ideal situations. At times the conveyor would speed up even though there were still retrievable parts, because the vision system (located upstream) did not find many part in the current image. At other times, the conveyor would slow when no parts were yet reachable due to the vision system's finding a lot of parts in the current frame with no allowance for the time required for the parts to get within reach of the robot.

Lastly, the system was highly dependant on the parameters used for the algorithm. Setting the maximum and minimum conveyor speeds too extremely amplified the non-collocation problem, while setting them too conservatively made the adjuster have little effect. The nominal conveyor speed and gain also effected the control by making it too reactive or too lethargic.

A second, modified control scheme was then tried. This algorithm still used the number of parts in the queue as the reference, but changed the rule used to compute a new conveyor speed. The algorithm was as follows: If there were more parts in the queue than a *high-water* mark, then the system would slow the conveyor by a set amount each time through the control loop ($\frac{1}{2}$ second intervals) until reaching a minimum speed. If there were less parts in the queue than a *low-water* mark, then the system would increase the speed by a set amount each time through the control loop until reaching a maximum speed. If the number of parts in the queue was between marks, then the system would *pull* the speed toward a median value by an amount determined using the current speed and median speed (using care to note current speed relative to the median).

```

if(current_speed > MEDIAN) {
    new_speed = current_speed - GAIN *
                (current_speed - MEDIAN)
} else {
    new_speed = current_speed + GAIN *
                (MEDIAN - current_speed)
}

```

Figure 8-2: Auto Adjusting Algorithm *Pull* Behavior

Figure 8-3 shows a graph of conveyor speed vs. time for an instance when this scheme was engaged (high-water mark→100; low-water mark→50; gain→0.10, step→5). While it behaved better the first algorithm, it was still quite sensitive to the setting of the various parameters.

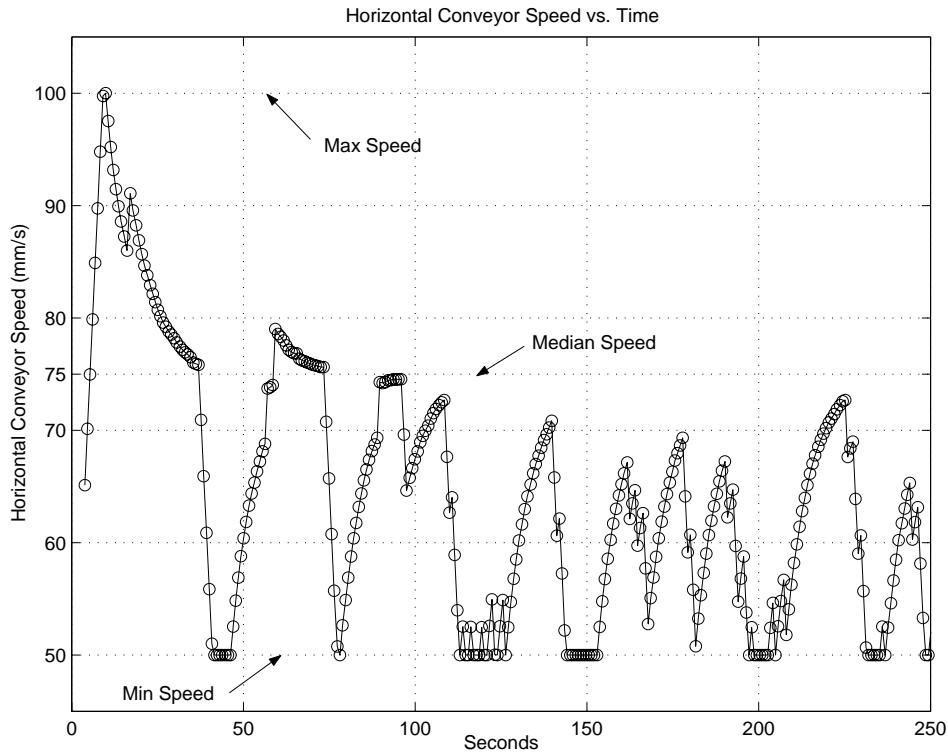


Figure 8-3: Auto-Adjusting Algorithm

While these two examples proved the auto-adjusting concept functionally works, there is clearly a need for further understanding and insight in creating adjusting algorithms. Since most of the feeder parameters (conveyor speed, camera setting, robot

setting, ...) are based in software, access to them for adjustment is trivial and has the potential to improve system throughput.

8.2 Future Work

Currently, an architecture for controlling flexible parts feeder has been proposed and implemented, yet there is much additional work to be done. Future work can be classified into three major areas: improving the functionality of the current system, extending the system to control a wider range of hardware components and feeder, and creating a knowledge base for *intelligent* algorithms capable of optimizing throughput using the part-specific auto adjuster object.

8.2.1 Improving Functionality

Several areas of the code could be expanded or further developed to improve overall function and design.

8.2.1.1 Error Handling

Currently, the software has some rudimentary error checking and handling through return values of method calls. An improvement would be the implementation of a hierarchical error-handling scheme [2] implemented through the try/catch construct [34], [35] available in C++. This would allow errors to be caught locally and analyzed. If it were a problem that could be handled locally, it could be remedied; otherwise, it could be passed to a higher level for attention. This would enable the feeder to be more robust to errors as they arise in the system. It would also help de-couple the error-handling code from the operational code.

8.2.1.2 Communications

A second area of improvement would be in a generic mechanism to handle all communications between system modules. Currently, a mixed bag of communications mechanisms is employed, including Ethernet sockets, home-grown message queues, Windows message passing schemes, and local method calls. Creating a generic communications proxy class and then constructing concrete implementations of that object for the various communication schemes would help to de-couple the actual communications from the objects that used them. Such class hierarchies have been implemented in other systems previously developed [3], [36], [37].

Additionally, the message queues should be unified to a generic queue. Currently, there are two separate but similar queues types in use in the system; one for passing text string messages and one for passing part objects (a part object holds all the information about a part which has been located by the vision system). A generic queue should be used to pass any type of object instead of using two specific queues.

8.2.1.3 Abstract Server-Level Base Classes

The server-level objects should be derived from abstract base classes which define generic interfaces to the hardware. This would simplify the construction of additional servers for controlling different pieces of hardware. Programming to the abstract interface would ensure that the new server was compatible with the rest of the system. An example would be the construction of a generic robot server class from which all robot specific objects could be derived [3].

8.2.1.4 Data Logging

Creating a general data logging object for the system would be beneficial for a number of reasons. First, a time-stamped record of system operations would be very useful when debugging the software. It is often very difficult to determine the cause of crashes when multiple threads are running asynchronously. Second, data collection during testing would be much easier if there were a single object through which to record data. For example, the data logging object could be instructed to write all *part_retrieved* events to a file for later analysis and plotting. Lastly, the auto-adjusting task would be simplified if there were a single source for all system statistics and data.

8.2.1.5 Multi-Part Queue

Currently, the parts queue is a single pipe. Parts are placed in one end and are retrieved from the other end in the same order in which they were inserted. However, there are times (e.g. when feeding multiple parts at once) that it would be advantageous to retrieve the next part of a *specified type* from the queue. For example, if a combination of washers and nuts were being fed, an assembly sequence might specify that first a washer be retrieved and then a nut. Currently, one would have to continue to remove nuts from the queue until a washer was found. There would then be the problem of what to do with the nuts pulled from the queue (while looking for a washer) but not retrieved.

To solve this problem, a multi-part queue could be developed. Externally, there would be a single `put()` and `get()` method, however, a second parameter to the `get()` method would indicate the desired part type. Internally, the new queue class would contain an array of FIFO's. Each individual part type would have its own FIFO. When a request to retrieve a certain part type was received, internally, the queue would go to the proper FIFO to get the part. If that part type was not available, the queue could either block waiting for that type of part to arrive, or could return with a failure.

The `put()` method would not have to change from its current signature since the `CPart` object already contains its part type as a data member. The new multi-part queue could simply consult the `CPart` object for its type and then place it into the appropriate internal FIFO. To maintain backwards compatibility with the current queue, the `get()` method could be overloaded such that a call to `get()` without specifying a part type would return a part from the first internal FIFO. When feeding a single part, all parts would be in the first FIFO by default. Figure 8-4 shows a graphic of the internal queue structure.

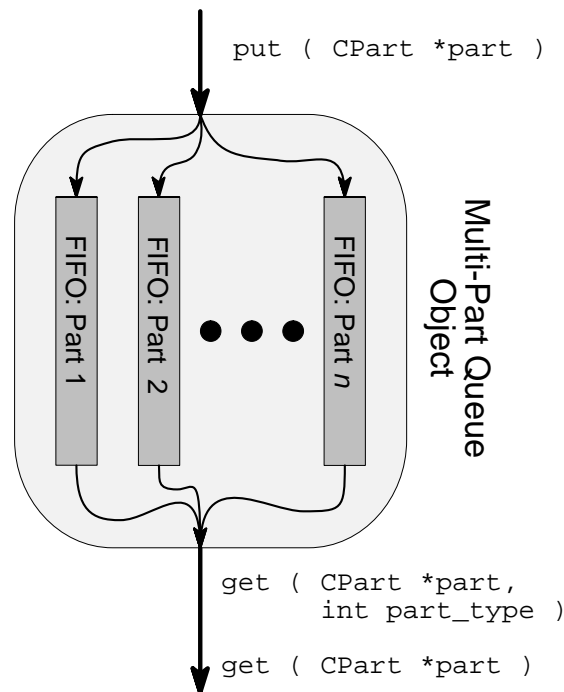


Figure 8-4: Multi-Part Queue

8.2.1.6 *MIL Encapsulation*

While a vision server object has been created which encapsulates much of the functionality of the MIL, much additional work remains. Currently, there are simply too many MIL calls littered throughout the locator PSOs. If the vision system were replicated with one from a different vendor, in addition to the vision server, the locator PSOs would also have to be modified.

Specifically, some basic functionality needs to be implemented. Generalized linear and circular rulers need to be developed. Additionally, a general image-frame-based transform functionality needs to be developed to make it easy to place vision recognition components in the image (e.g. starting at the centroid, extend a ruler at a 90° angle to the blob's major axis to find the blob's edge). This would also make it easy to draw lines on the image as a visual aid to vision-recognition routine debugging.

8.2.1.7 *Utility Programs*

In addition to the main feeder control program (which was the main focus of this thesis) several utility programs are needed to operate the feeder. They include a program for calibrating the robot relative the conveyor and calibrating the robot relative to the camera. While these procedures need only be done when the system is constructed or physically modified (after setup, the calibration values are stored on disk and retrieved at power-up), they are an important step in preparing the feeder for operation. Currently, simple, console-based programs have been written to perform these functions. However, more elaborate versions utilizing a GUI for operation would be easier for the average operator to use.

In addition to conveyor↔robot and camera↔robot calibrations, tool offsets must be determined for each gripper used in the system. Currently, a manual procedure is followed to determine the offset, however, a simple program which guided the user through a series of steps and then computed the offset would make the process easier and less error prone.

8.2.1.8 *Remote User GUI*

As mentioned in Chapter 4, the remote user GUI, while functional, is not a well designed piece of code. There are many opportunities for improvement in its design and

coding. Currently, much of the functionality resides in a single source file with many global variables. Segmenting and modularizing the program (and source files) would make future improvements and modification much easier. Additionally, the source contains very little comments that could otherwise make it easier to understand at a later date.

8.2.1.9 Variable / File Naming Convention

Lastly, as the main feeder control code has been developed over the past two years, a variety of variable-naming and file-naming conventions have been used. There is a combination of all lower-case variable names composed of words joined by underscores, mixed-case names composed of words butted together with first letters capitalized, and combinations of single-word and multi-word variables. A uniform naming convention for variables throughout the complete code base would make later revision and update much easier.

Secondly, source code file names have suffered from a similar situation. A mixture of various names have been used. This makes it difficult to quickly locate a particular source code file when wanting to update a class method or further determine the exact implementation of a function. A uniformed naming scheme would enable quicker access to desired source code and would make it easier to understand the functionality implemented in a particular source file.

8.2.2 Extending Capabilities

In contrast to improving the functionality of the system, work also needs to be done in applying the software architecture to a wider range of hardware. This could entail many different scenarios. The feeder control code could be used to replace the current control code used on the original CWRU feeder. The would examine how the software would handle a different way of operating the system (serial sub-system operation vs. parallel). A second feeder could be constructed using different hardware. Perhaps the robot could be replaced with one from a different manufacturer, or the current conveyor configuration could be replaced with a series of vibrational elements or the part presentation sub-system from Adept's FlexFeeder 250. This should be

accomplished through writing new server-level objects, the rest of the software should remain the same.

The goal of extending the capabilities of the system would be to better understand how various hardware components from different manufacturers would affect the overall system. The design philosophy for the software architecture has been to encapsulate manufacturer-specific behaviors behind generic interfaces, however, until one faces the challenges of using different components in the same capacity, the level of de-coupling is not truly known.

8.2.3 Intelligent Auto-Adjustment

Another area for future work is in the development of intelligent algorithms for optimizing system throughput. This is a very large research area in which little progress has been made. Some small results have appeared in the literature [38], [39], [40], [41], [42], however, much is still to be done.

Ideally, one envisions an object that would periodically wake up, examine the current state of the system, fuse that new data with an existing knowledge base, and adjust feeder parameters to improve system operation. The type of data that could be used in making such a decision would include not only the current state of the system, but also other items, such as: the past performance of the system, statistical data from the overall operation of the feeder, and models of the system that could predict what the throughput should be, in the ideal case.

Currently (as clear from the results section), the important parameters to monitor and adjust are not known. A base of knowledge needs to be built up which would recommend guidelines for adjustment algorithms for various feeding situations. Consider a hypothetical example; if it were known that when feeding torus-shaped parts, a good monitor of overall throughput was number of parts located but not retrieved, then when developing an algorithm for feeding washers, one would have a head start on generating a good auto-adjusting scheme.

An additional adjustment option which was briefly examined was in the intelligent manipulation of the parts queue. It was suggested that the time required to retrieve any given part might be dependant on its physical location on the conveyor.

Thus, by ensuring that the robot retrieve the better-positioned parts first, overall throughput could be improved. To check this, data was gathered and a plot was generated (Figure 8-5) showing retrieval time (expressed in parts per minute) vs. location on the conveyor from which the part was retrieved (when feeding lenses). While nothing definitive was concluded from this endeavor, it is something that should be examined in further detail in the context of auto-adjustment.

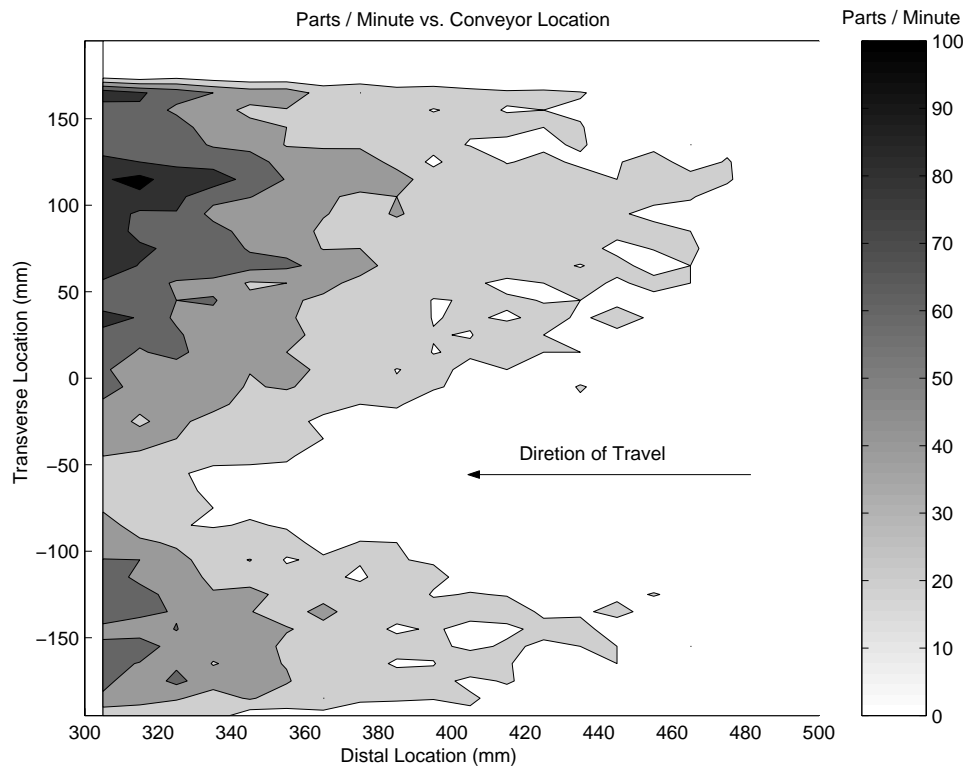


Figure 8-5: Retrieval Time vs. Part Location

8.3 Conclusions

An object-oriented software architecture has been developed and implemented for the new CWRU flexible parts feeder. It has been shown capable of controlling the new feeder and has the capacity to change the current part-feeding scenario (via user selection) without shutting down and restarting the system. This was accomplished by encapsulating part specific code behind well-defined interfaces and using the abstract factory design pattern to construct part-specific objects as required.

The system has successfully fed two different parts in three different scenarios and has been shown to feed 2 inch diameter flashlight lenses at speeds in excess of 60 parts per minute. This represents a 6X improvement in throughput over the original CWRU feeder design (for the exact same part).

The system, as it currently stands, is best considered a foundation on which further research into the control of flexible parts feeding may be launched. As was clearly outlined in the previous sections, there are many areas in which further research may be directed. These include not only the further design and refinement of the software architecture, but the greater problem of fundamentally understanding and controlling the underlying dynamics and mechanisms which cause flexible feeding systems to behave as they do.

9 Bibliography

- [1] G. Causey *et al.* Testing and Analysis of a Flexible Feeding System. *Proc. IEEE International Conf. Robotics and Automation*. Detroit, MI, April 1999.
- [2] Y. Kim *et al.* A Flexible Software Architecture for Agile Manufacturing. *Proc IEEE International Conf. Robotics and Automation*. Albuquerque, NM, April 1997.
- [3] J. Swaminathan. *An Object-Oriented Application Framework for Peg-In-Maze Assemblies*. M.S. Thesis, Dept. of Electrical Engineering & Computer Science, Case Western Reserve University, January 2000.
- [4] Adept Technology Inc., "Adept FlexFeeder 250," San Jose, CA 95134, (408)-432-0888, http://www.adept.com/products/flex_feeder
- [5] A. Pedrazza. Performance Strategies for Flexible Parts Feeding. *IEEE International Conf. Robotics and Automation. Proc. Flexible Parts Feeding Workshop (W5)*. San Francisco, CA, April 2000.
- [6] Seiko Instruments USA, Inc., Factory Automation Division, Torrance, CA 90505, (310)-517-7850, <http://www.seikorobots.com/>
- [7] Hoppmann, Haymarket, VA 20169, (800)-368-3582, http://www.hoppmann.com/FLEX_S~1.HTM, <http://www.hoppmann.com/fw4500.htm>
- [8] S. J. Gordon, Flexible Feeding of Small Parts, *Presented at the 1996 RIA Flexible Parts Feeding for Automated Handling and Assembly Workshop*, Minneapolis Marriott City Center, Minneapolis, MN, Oct 1996.
- [9] E.M. Ross. Flexible Feeding Devices for Robotic Assembly. *Presented at the 1994 RIA Flexible Parts Feeding for Automated Handling and Assembly Workshop*, Westin Hotel, Cincinnati, OH, Oct 1994.
- [10] F. L. Merat *et al.* Design of an Agile Manufacturing Workcell for Light Mechanical Assembly. *Proc. IEEE International Conf. Robotics and Automation*. Minneapolis, MN, April 1996.
- [11] G. Causey *et al.* Design of a Flexible Parts Feeding System. *Proc IEEE International Conf. Robotics and Automation*. Albuquerque, NM, April 1997.
- [12] G. C. Causey. *Elements of Agility in Manufacturing*. Ph.D. Dissertation, Dept. of Mechanical & Aerospace Engineering, Case Western Reserve University, January 1999. (Chapter 2)
- [13] G. Causey. Design, Testing, Modeling and Control of Flexible Parts Feeding Systems. *IEEE International Conf. Robotics and Automation. Proc. Flexible Parts Feeding Workshop (W5)*. San Francisco, CA, April 2000.
- [14] W. Wolfson and S. Gordon. Designing a parts feeding system for maximum

- flexibility. *Assembly Automation*, Vol. 17, No. 2, pp. 116-121, 1997.
- [15] Adept FlexFeeder 250 User's Guide. Rev. B. (Nov. 1998), Adept Technology Inc., San Jose, CA 95134, (408)-432-0888, <http://www.adept.com>.
- [16] Cimetric Inc., Salt Lake City, UT. <http://www.cimetric.com/>
- [17] S. Sorenson and R. Stringham. Vision guided flexible feeding made easy. *Assembly Automation*, Vol. 26, No. 2, pp. 99-104, 1999.
- [18] Doxygen: documentation system for C++, Java, IDL, and C. © 1997-2001 by Dimitri van Heesch. <http://www.doxygen.org/>
- [19] G. Causey. "High-Speed Flexible Parts Feeder: Software Reference Manual", Center for Automation and Intelligent Systems Research, Technical Report 01-004, CWRU, 2001.
- [20] G. Causey. "High-Speed Flexible Parts Feeder: Source Code", Center for Automation and Intelligent Systems Research, Technical Report 01-005, CWRU, 2001.
- [21] Adept MV Controller User's Guide for V+ Version 13.0 or Later Rev. B. (1999), Adept Technology Inc., San Jose, CA 95134, (408)-432-0888, <http://www.adept.com>.
- [22] Adept Windows User's Guide Version 2.1 (1999), Adept Technology Inc., San Jose, CA 95134, (408)-432-0888, <http://www.adept.com>.
- [23] Galil Motor Control, Inc., Mountain View, CA 94043, (800)-377-6329, <http://www.galilmc.com>.
- [24] Matrox Imaging, Matrox Electronic Systems, Ltd. Dorval, Quebec H9P 2T4, Canada, (800)-804-6243. <http://www.matrox.com/>
http://www.matrox.com/imaging/products/meteor2_mc/home.cfm
- [25] Matrox Imaging, Matrox Electronic Systems, Ltd. Dorval, Quebec H9P 2T4, Canada, (800)-804-6243. <http://www.matrox.com/>
<http://www.matrox.com/imaging/products/mil/home.cfm>
- [26] E. Gamme *et al.* *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994
a) Abstract Factory, pp. 87.
b) Singleton, pp. 127.
c) Proxy, pp. 207.
d) Adapter, pp. 139.
- [27] D.C. Schmidt. *Reality Check C++ Reports*, SIGS, Vol. 8, No. 3, March 1996.
- [28] T. Pham and P. Garg. *Multithreaded Programming with Win32*, Prentice Hall, 1999.
- [29] DMC-18X2 Command Reference Manual Rev. 1.0b, Galil Motor Control, Inc.,

Mountain View, CA 94043, (800)-377-6329, <http://www.galilmc.com>.

- [30] R. Eckstein, *et al. Java Swing*, O'Reilly, 1998.
- [31] Sun On-line Java Documentation, Sun Microsystems, Palo Alto, CA, 94303.
<http://java.sun.com/j2se/1.4/docs/api/javaw/swing/SwingUtilities.html>
- [32] AdeptNet User's Guide Rev. A. (1996), Adept Technology Inc., San Jose, CA 95134, (408)-432-0888, <http://www.adept.com>.
- [33] V+ Operating System User's Guide Ver. 13.0 (1998), p. 99, Adept Technology Inc., San Jose, CA 95134, (408)-432-0888, <http://www.adept.com>.
- [34] S. Lippman and J. Lajoie. *C++ Primer 3rd Ed.*, Addison Wesley, 1998.
- [35] S. Oualline. *Practical C++ Programming*, O'Reilly, 1997.
- [36] N. A. Barendt *et al.* A Distributed, Object-Oriented Architecture for Platform-Independent Machine Vision, *Proc. IASTED Conference on Robotics and Manufacturing*, 1998.
- [37] N. A. Barendt. *A Distributed, Object-Oriented Architecture for Platform-Independent Machine Vision*, M.S. Thesis, Dept. of Electrical Engineering & Applied Physics, Case Western Reserve University, May 1998.
- [38] B. Mirtich *et al.* Estimating pose statistics for robotic parts feeders. *Proc. IEEE International Conf. Robotics and Automation*. Minneapolis, MN, April 1996.
- [39] D. Gudmundsson and K. Goldberg. Tuning robotic part feeder parameters to maximize throughput. *Proc. IEEE International Conf. Robotics and Automation*. Albuquerque, NM, April 1997.
- [40] D. Berkowitz and J. Canny. Designing parts feeders using dynamic simulation. *Proc. IEEE International Conf. Robotics and Automation*. Minneapolis, MN, April 1996.
- [41] J. Craig. Simulation-based robot cell design in AdeptRapid. *Proc. IEEE International Conf. Robotics and Automation*. Albuquerque, NM, April 1997.
- [42] K. Goldberg *et al.* Estimating throughput for a flexible parts feeder. *Proc. International Symp. Experimental Robotics*. June 1995.