Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 271 **April 1985**

Using the PUMA System

Joe L. Jones

Patrick A. O'Donnell

This document describes the operation of the Lisp Machine interface to the Unimation Puma 600 Robot Arm. The interface is evolved from a system described in an earlier paper, and much is the same. However, the under-lying interface between the Lisp Machine and the Puma has changed and some enhancements have been made. VAL has been replaced with a PDP-11/23, communicating with the Lisp Machine over the Chaosnet.

The purpose of this document is to provide instruction and information in the programming of the Puma arm from the Lisp Machine. The network protocol is not described here, nor are the internals of the implementation. These details are provided in separate documentation.

The reader will find in this paper both a tutorial section and a reference section. The tutorial will lead the reader through a sample session using the Puma by directly calling the primitive operations, and will provide an introduction to programming using the primitives. The reference section provides an overview of the network protocol and describes all of the primitive operations provided.

Please note that this document corresponds to the version of the Puma system in use on 11 March, 1985. The system is still undergoing development and enhancement, and there may be additional features, if you are running a newer system. The authors welcome reports of errors, inaccuracies, or suggestions for clarification or improvement in either the documentation or the code for the Puma system. Please send electronic mail to BUG-PUMA@MIT-OZ.

Contents

1 Introduction: The Basics

1.1 Structure of the System

The Puma arm is controlled primarily by a PDP-11/23, called MIT-PUMA. (See Figure 1.) It controls the motion of the arm through a set of six microcomputers, one for each joint. It controls the arm in response to commands from the Lisp Machine, telling it where to move the arm, and asking for information, such as where the arm currently is. The Puma PDP-11 is assisted in its task by another PDP-11/23, MIT-COUGAR, which monitors strain gauges in the wrist of the arm, and suggests modifcations to the trajectory calculated by the Puma PDP-11.

The Lisp Machine issues commands to the Puma PDP-11 by sending request packets over the network. Each request packet contains one command for the Puma, such as "tell me your current position," "set your speed factor to 0.4," or "add this segment to the trajectory." (The latter example will be elaborated upon shortly.) Every request expects a response, which may contain information asked for by the request, simply a confirmation that the request was received and acted upon, or an error message indicating that there was some reason why the request could not be satisfied.

As described in more detail later, there are several queues on which a request or response may be stored while awaiting processing. When a user program creates a request, and asks the Puma PDP-11 to perform, it may or may not wait for the response. The request may be queued, and the response looked for later, if at all. However, the Lisp Machine will always expect a response from the Puma PDP-11, and the PDP-11 will always supply one for each request.

The meaning of a response will vary, depending on the request, but every response indicates that whatever action had been requested has been acted upon. In the case of motion requests, however, "acted upon" may be a little unintuitive. The Puma system is designed so that several set points may be sent to the Puma PDP-11, which will move the arm to each of them, with smooth transitions between each command. The motion from one set point to the next is referred to as a *segment*. If the Lisp Machine had to wait for the arm to finish one segment before requesting the next, there would be no way for the Puma to calculate a smooth transition from one segment to the next. Thus, the Lisp Machine can send several segments to the Puma which then puts them on a queue to be executed in order. The response that the Puma returns to the Lisp Machine in this case means "I queued your request, and I will perform that motion when all preceeding motion has been performed." This behavior is important to understand, as it has several consequences which will be explored in later sections of this document.

1.2 The Arm

The Puma arm has six revolute joints. (See Figure 2.) They are numbered starting from the base with joint 1. They are also anthropomorphically referred to as (again starting from the base) the trunk, the shoulder, the elbow, and the wrist (for the last three joints).

The position of the arm corresponding to all joints being zero is as shown in the figure. The second link (the upper arm) is horizontal, pointing toward the outside wall of the building, and is on the side of the trunk toward the room. The third link (the forearm) is vertical. The joint 5 axis is pointing out into the room. The fingers are pointing toward the ceiling, and would open toward the sides of the room. It is like the left arm of a person facing the Unimation controller, contemplating his forearm or palm. *Note that this position is* not *the same as the "ready" position, with the whole arm pointing straight up.*

Assume the arm is in the position described in the previous paragraph. Positive rotation for all the joints is counterclockwise, either looking down on the robot from above, or looking at the robot from the Lisp machine console. Note that this means that positive angles for joints 2 and 3 (in this "lefty" configuration) are *down toward the table! Be careful!* (In the "righty" configuration, where joint 2 is flipped over to the

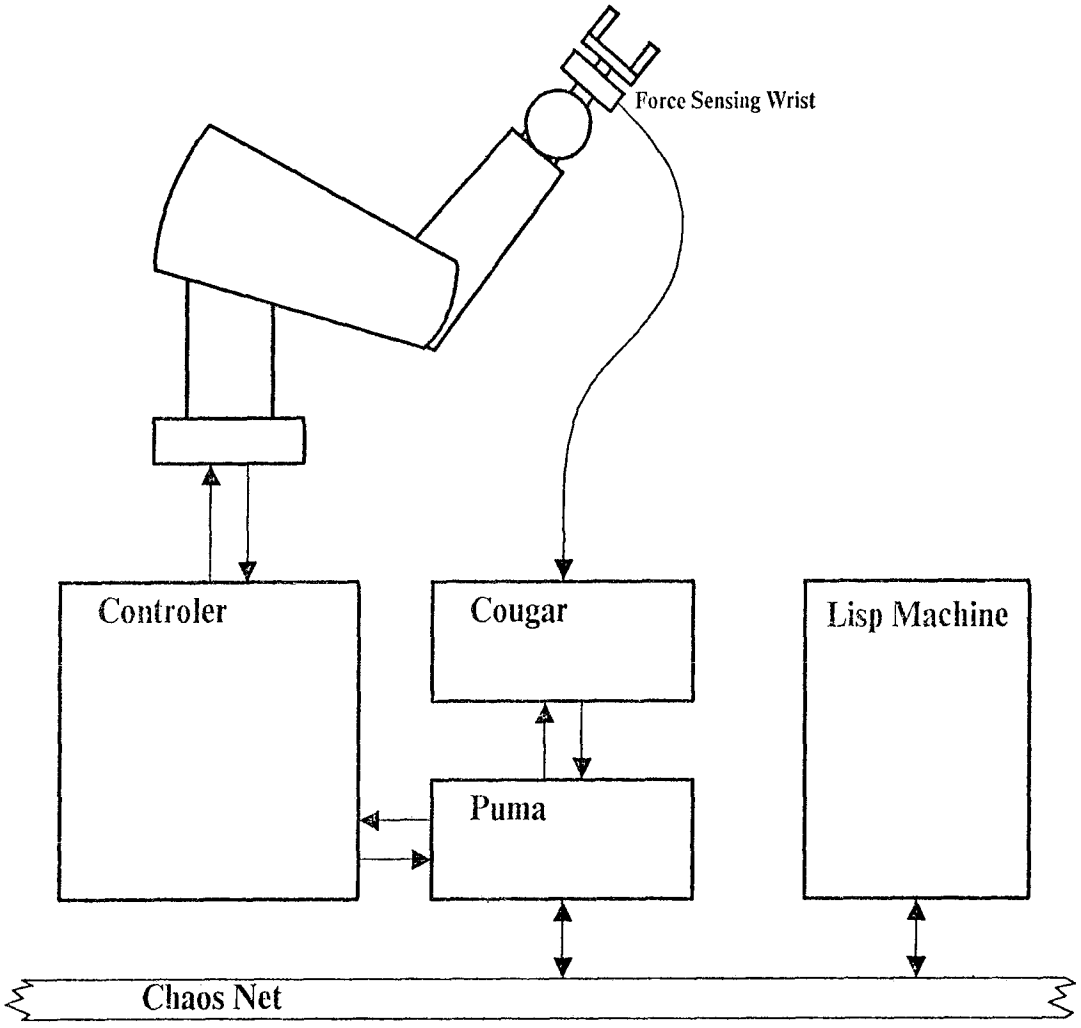

Figure 1: Block diagram of the Puma System

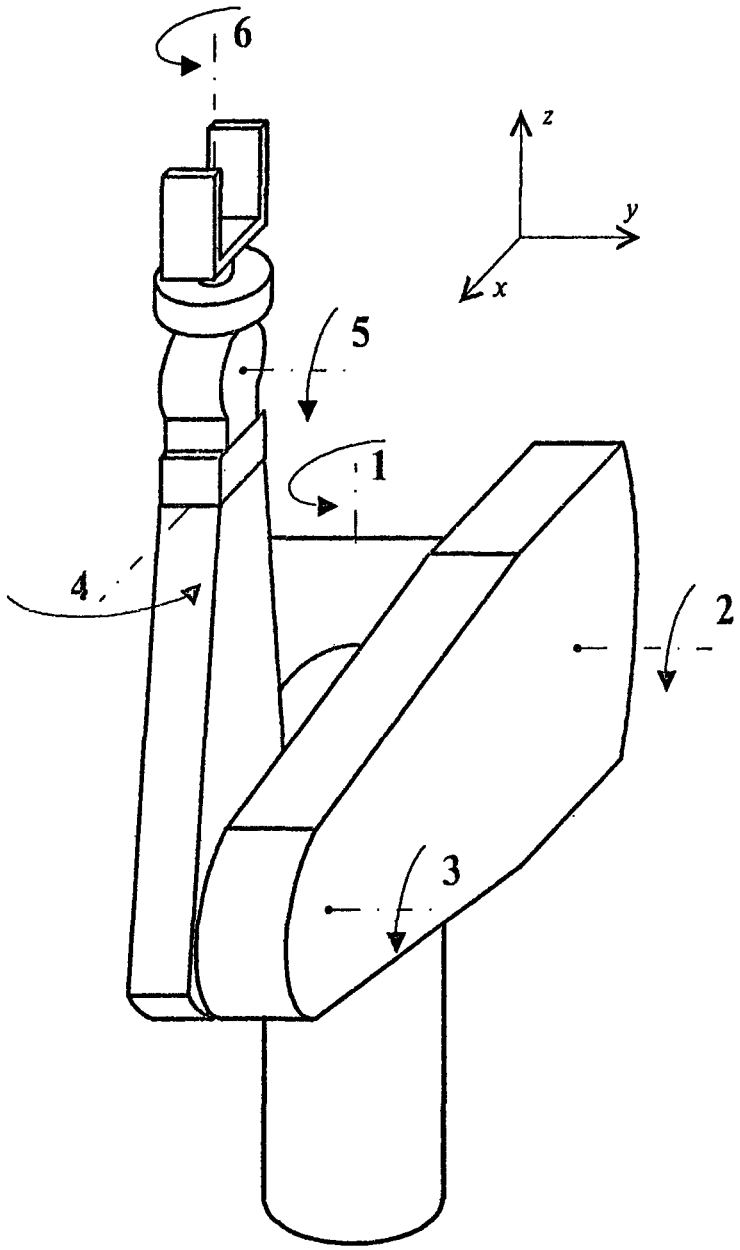

Figure 2: Direction of motion for positive changes in joint angles. The x, y, and z directions are also shown.

other side of the trunk, positive rotation would be away from the table.) Positive motion of joint 1 from the zero position would bring the arm to point out into the room. Positive rotation of joints 4 and 6 would be that which would drive a normally threaded screw or bolt.

For cartesian positioning, the "world" coordinate system is defined as follows (see also Figure 2): The origin is at the intersection of the axes of joints 1 and 2. This is in the middle of the "shoulder" of the arm. The z axis points straight up, along the joint 1 axis. The y axis points out into the room (along joint 2 axis, when the arm is in the zero position). The positive x axis is in the direction of the Unimation controller.

1.3 Talking to the Puma

The Puma Arm is represented in the Lisp world by an instance of the puma flavor. Instructions are given to the arm by sending messages to that instance. The Puma object then sends the appropriate command packets to the PDP-11. The defined messages which may be sent to the Puma object are listed in Section 6, page 30.

Each message that the Puma instance accepts may send one or more request packets to the Puma PDP-11. Some of the operation will wait for a response—those which expect some data to be returned, and some will only wait if instructed to. See the reference section for more details. Note again, though, that any waiting the motion operations may do does not mean that they wait until the motion has actually been performed. It only means that the request has been queued to the motion segment queue.

1.4 What to do about errors

Occasionally, the Puma PDP-11 may signal an error condition back to the Lisp Machine. The handling of the network connection is performed in a background process, which attempts to find an appropriate place to send the error. This will frequently be the user process, in the case of operations which wait for completion. However, if the completion of the request is not being waited on, then an error will be signalled in the background process, and a notification will be sent to the user. In that case, no further operations can be processed until the error is handled by the user. The following procedure, or an equivalent, should be followed:

- *Select the background lisp interactor by typing* `<Function> 0 S`, *as instructed by the notification.*
- *Examine the error message, to see if any action needs to be taken (such as turning arm power on), and perform the actions necessary.*
- *Proceed from the error by hitting the* `<Resume>` *key.*

Certain errors are not serious, and do not require direct intervention by the user before proceeding. It is possible to set the Puma flavor instance to merely send a notification for these errors rather than entering the debugger. This may be set when initializing the connection (as described in the reference section under the `:initialize` operation), or by sending a `:set-notify-errors?` message (see the reference section).

Part I

A Tutorial Introduction to the Puma System

2 Tutorial

This tutorial will lead the reader through the steps necessary to start up, operate, and shut down the Puma system. Simple operations are presented, sufficient to begin programming the Puma by specifying positions to which to move, how to move there, and operating the hand. The next chapter will present more advanced features such as compliance and guarded motion.

The instructions and examples in this section are designed and presented in such a way that the reader may follow along at the Lisp Machine console, following the instructions and trying out the examples. In fact, it is recommended that the reader do so, as hands-on experience is the best teaching method.

Instructions in this tutorial describe the sequence of operations required to achieve a goal. They are presented in a *slanted typeface*, indented, and itemized. Lisp code is presented in a `typewriter typeface`, with generic arguments for functions in *slanted typeface*. Example:

- *Go to a Lisp Machine from which you can communicate with the arm. (I.e. Robot-2).*
- *Log in to the Lisp Machine. Evaluate* `(login username)`.
- *Proceed with reading this tutorial.*

Numerous examples are presented here. They are all presented in the `typewriter typeface`. Forms the *user* types are underlined. Comments—lines preceeded by semicolons—are explanatory and, of course, do not actually appear when the example is executed. All of the examples in this document were actually executed, and are thus guaranteed to work in the version of the system existing on 11 March, 1985. An example of an example:

```
;;; Logging in to the machine.
(login 'user)
Loading OZ:PS:<USER>LISPM-INIT.BIN into package USER
T
```

A note on the use of *puma*: All the examples in this document use the global variable *puma* where a Puma instance is stored for the purposes of this tutorial. The instructions, however specify *puma* as the place to send the message. In user programs, of course, *puma* would have the program's name for the puma instance substituted.

2.1 Starting up the system

In order to start up the Puma system, we need to load the Lisp software into the Lisp Machine we are using, we need to power on the PDP-11's and Unimation controller, and we need to establish the Chaosnet connection to the Puma.

2.1.1 Loading the Lisp software

In theory, the system may be operated from any computer connected to the Chaosnet. In fact, the arm has been operated from both Lisp Machines and a VAX. However, for practicality and safety, the PDP-11 will only accept connections from computers with consoles physically in the room with the robot. At the time of this writing, these computers are CADR-25, Robot-2, and Robot-3.

Robot-2 customarily has a local world load with the latest Puma system pre-loaded. To use the Puma on that machine, we simply boot that world load:

- *Get to the FEP, by either evaluating* `(si:halt)` *or hitting* `Hyper-Control-Function`.
- *Type* `boot >puma.boot`.

To load the Puma system for use on CADR-25 or Robot-3, or on Robot-2 if the provided world is unsatisfactory:

- *Evaluate* `(make-system 'manip :noconfirm)` *followed by* `(make-system 'puma :noconfirm)`. *The* `manip` *system must be made before the* puma *system.*

[Note: All development of the Puma system takes place on the Symbolics 3600. Periodically, the system is compiled and tested on the CADR to maintain compatibility. The effect is that the CADR version of the system is may not be up-to-date. Contact the maintainers if there is any problem.]

2.1.2 Powering on the PDP-11's

There are two PDP-11/23's used in the Puma system. One, named MIT-COUGAR, is used for all force measurements and compliant control. If you will be using neither of these features, then that machine need not be powered on.

- *Power on the Unimation controller: Flip the toggle switch on the front panel of the controller to the "ON" position (pointing up). [Note: It doesn't matter what position the black knob is in—it is not used, nor is the button labeled "Auto Start".]*
- *Power on the PDP-11's: Make sure all the white switches on the front panels of the computers are in the right positions—the "HALT" switch should be set to "ENABLE", and the "LTC" switch should be "OFF" for MIT-COUGAR(the upper one), and "ON" for MIT-PUMA. Flip the red toggle switches to "ON". It doesn't matter in which order this is done.*
- *Note: The red "RUN" lights on the PDP-11's should be on at this point. If they are not, simply toggle the "BOOT" switch up on the offending processor, and they should boot up.*

The Unimation controller requires the Unibus INIT signal generated by the MIT-PUMA on startup, and so must be powered on before MIT-PUMA.

Note: Arm power is not turned on yet. The PDP-11 doesn't enable it until the network connection is established.

2.1.3 Creating a Puma instance

We must have an instance of the puma flavor to which to send messages. We create one with the `make-instance` function. Of course, we need some place to keep it, so we set a variable to it.

- *Evaluate* `(setq *puma* (make-instance 'puma:puma))`.

2.1.4 Initializing the network connection

Before any operation of the arm can proceed, the network connection between the Lisp Machine and the PDP-11 must be established:

- *Evaluate* `(send puma :initialize)`.

The `:initialize` operation will query the user to make sure the PDP-11 and the controller is turned on. If this is not desired, the operation accepts an optional argument which, if `T`, will suppress this query. See the reference information for additional arguments.

Example:

```
(send *puma* :initialize)
Make sure that the Puma 11/23 is turned on and has booted,
that it is connected to the Puma controller, and that the
Puma controller power is turned on.
Proceed? (Y or N) Yes.
Initialization successfully completed.
T
```

At this point, the software is ready to operate the arm. It is now possible to turn on the power to the arm. No arm motion is possible until the high power is on. Note: it is recommended that the red panic button be kept within easy reach while the arm power is enabled. One never knows...

- *Press the black "Arm Power On" button on the panel of the Unimation controller. Keep one hand near the red "Arm Power Off" button, should the arm begin to move when power is turned on.* The arm should not move! *(With the possible exception of a small jerk.) If it does, turn off power* immediately, *and contact a maintainer.*

The red indicator light above the button should come on and remain on. If the light does not remain on, then the key switch in the remote panic button may not be turned on. Press the key down, and turn counter-clockwise until it stays down.

2.1.5 Calibrating the arm

Calibration of the arm is necessary to ensure accurate repositioning to stored setpoints. For best results, the arm should be calibrated each time the arm power is turned on. Theoretically, it is only necessary to recalibrate when powering on the Unimation controller and Puma PDP-11. However, it has been observed that under some circumstances of shutting arm power off (e.g. hitting the panic button, emergency stop for collisions), the microprocessors in the controller can lose track of some encoder counts. Recalibration each time the arm power is turned on is the safest, most reliable, procedure.

- *Evaluate* `(send puma :calibrate)`.

Each joint in succession will move a small amount. The `:calibrate` operation will return immediately, leaving the PDP-11 to perform the calibration operation. If you wish to have the operation wait until calibration is actually completed, add the optional keyword argument of `:wait-p t`.

Example:

```
(send *puma* :calibrate)
T       ; The arm moves even after the form returns.
```

2.1.6 Summary

In summary, to start up the Puma system,

- *Load the Puma Lisp system on the Lisp Machine (or boot to the correct world load).*
- *Power on and bootstrap the Unimation controller and the PDP-11's.*
- *Create the* puma *flavor instance:* `(make-instance 'puma:puma)`.
- *Establish the network connection:* `(send puma :initialize)`.
- *Turn on arm power.*
- *Calibrate the arm:* `(send puma :calibrate)`.

The arm is now ready for operation.

2.2 Getting information about the arm

In this section we explore how we can ask the arm to report its current position, in joint angles and in Cartesian coordinates. A few other information gathering operations are also presented.

2.2.1 Joint angles

To read the current joint angles,

- *Evaluate* `(send puma :joint-angles)`.

This form will return an array of six floating point numbers representing the joint angles for the arm.

Example:

```
(setq some-angles (send *puma* :joint-angles))
#<ART-Q-6. 27402731>
(listarray some-angles)
(-2.75414 -178.50954 179.94504 1.0984688 -1.8171648 -1.1155959)
```

2.2.2 Transforms

We can ask the Puma for it position in Cartesian coordinates. It returns a 4 by 4 transform matrix representing the position and orientation of the hand. (A full discussion of transforms is beyond the scope of this document. For a good introduction, see *Robot Manipulators: Mathematics, Programming, and Control*, by Richard P. Paul, 1981, MIT Press, Cambridge.) Some details of transform implementation in the Puma system may be found in the reference section.

The point which is represented by the transform is the intersection of the axes for joints 4, 5, and 6. Thus, changing the hand orientation in the transform will not move joints 1, 2, or 3 (unless, of course, the configuration needs to be changed due to the joint limits). The origin of the world coordinate system is located at the intersection of the axes of joints 1 and 2. The positive z-axis points up, the x-axis points toward wall with the windows, and the y-axis points out into the room.

To get the transform from the Puma,

- *Evaluate* (send *puma* :here).

Example:

```
(setq here-i-am (send *puma* :here))
#<Transform Here (#30) 27463222>

(manip:transform-print here-i-am)
 1.00    0.05   -0.01    -433.50
-0.05    1.00   -0.00     170.95
 0.01    0.00    1.00     444.68
 0.00    0.00    0.00       1.00
NIL
```

The units of distance used by the transformations are millimeters.

2.2.3 Other

Other more or less useful data may be retrieved from the Puma. For example, a "description" of the manipulator is returned by the :describe-manipulator operation:

```
(send *puma* :describe-manipulator)
(:BRAND-NAME "Unimation Puma 600" :UNIQUE-NAME "Puma"
:SOFTWARE-VERSION (:PMVERS 515. :UTVERS 194.  :SLVERS 41.
:ARVERS 17. :MVVERS 201. :CLVERS 102.  :CPVERS 235.
:DAY 28. :MONTH 1. :YEAR 85.) :PROTOCOL-VERSION 1.
:DEGREES-OF-FREEDOM 6.  :OPERATIONS (128. 129. 130. 131.
132. 136. 139.  140. 141. 142. 143. 144. 145. 146. 147. 148.
149.  150. 151. 152. 153. 160. 161. 162. 163. 164. 165.
166. 167. 168. 169. 170. 171.))
```

It is sometimes (but, admittedly not often) useful to know what version of software the PDP-11 may be running. As the system is developed, occasionally new features may be added, and incompatibilities crop up. The current version may be displayed as follows:

```
(send *puma* :describe-version)
Module  Version
 PUMA      515
 PMUTIL    194
 PSOLN      41
 ARITH      17
 MVPROC    201
 CALINF    102
 COMPLY    235
Assembled on Monday, 28 January, 1985
NIL
```

There are several parameters which can be examined by the :get-parameter operation. The operation takes one keyword argument indicating what parameter you wish to examine. For example:

```
(send *puma* :get-parameter :speed-factor)
1.0
(send *puma* :get-parameter :interrupts-missed)
0.
(send *puma* :get-parameter :transition-time)
4.
```

2.3 Moving the arm around

We now can actually move the arm. First, however, we should see just how to make sure the arm doesn't do any damage to itself or anything around it, should it go somewhere we didn't quite expect. Then we will examine how to specify joint angles for the arm, how to specify positions and orientations in Cartesian space, and straight line motion.

2.3.1 Emergency stop—the panic button

At any time when the arm is moving (and even when it's not), power to the joint motors may be cut off by hitting one of the two red panic buttons connected to the controller. One is on the controller panel itself, labelled "ARM POWER OFF". The other is on a remote box, connected to the controller through a white cord.

Pressing either panic button will immediately cut off power to the joint motors and engage the brakes on the large joints (joints 1 to 3). This should effectively stop the arm. The PDP-11 will then notice that the power is off, and, if the arm was moving, notify the Lisp Machine, causing an error condition or notification there.

Whenever you are moving the arm it is wise to have the remote panic button near your hand, and, in fact, have your hand hovering over the button. It is especially wise when testing new programs.

To turn power back on, follow the instructions in Section 2.1.4, page 10. If the arm had been moving, the following will also be necessary.

- *Evaluate the form* (send *puma* :reset-abort-condition).

See the reference section under the operation :reset-abort-condition (Section 6.3, page 34) for a discussion of why this is necessary.

Under certain circumstances, the PDP-11 may shut the arm off itself, for instance when it discovers that the user didn't have the panic button nearby and the arm just slammed into the table—probably damaging the hand, the table, the gears, the bearings, and burning out a few motors. When the PDP-11 turns off the power in this way, simply pressing the "ARM POWER" button will not reestablish power. First, the :enable-arm-power message must be sent to the Puma, presumably after repairing whatever damage had been done. Of course, there are other considerations to recovering from a situation such as this, including the fact that the enagement of the brakes on joints 1, 2, and 3 has maintained a tension between the arm and the table, which may cause the PDP-11 to immediately decide to kill the power again. By far, the best solution is to not let the arm get into this position.

A slightly less drastic method may be used to stop the arm quickly under program control. The :abort-move operation will instruct the PDP-11 to immediately stop the arm and forget about all pending move segments (see below). The arm is *not* gracefully decelerated, so this operation should be used only when necessary, and then with care.

2.3.2 Setting the joint angles

Now that we know how to stop the arm, let us examine how to start the arm. One simple way is to specify the value for each of the joint angles. First, however, let us put the arm in the "ready" position, which is pointing straight up:

- *Evaluate* (send *puma* :ready).

Notice that the (send *puma* :ready) form returns right away, before the arm motion is completed. This is because the operation merely queues the destination on the PDP-11, then leaves the PDP-11 to move the arm while the Lisp Machine can continue its calculations. Several motion commands may be sent to the Puma, and they will be executed in sequence, with a smooth transition between each segment.

To specify the joint angles for the arm, simply execute the :set-joint-angles operation:

- *Evaluate* (send *puma* :set-joint-angles *joint-angles*).

The *joint-angles* argument may be either a list or a one dimensional array, containing the six joint angles. For example,

```
(send *puma* :set-joint-angles '(0 0 0 0 0 0))
T

(setq test-angles (fillarray (make-array 6) '(0 -90 90 0 0 0)))
#<ART-Q-6. 34551264>

(send *puma* :set-joint-angles test-angles)
T
```

(As can be seen from trying this example, the joint angles 0, -90, 90, 0, 0, 0, correspond to the ready position.)

Frequently, we may want to suspend processing on the Lisp Machine until the arm has completed some sequence of motions. We can do this by executing the :wait-for-move operation, which will issue a request to the PDP-11 which will not complete until all queued motion requests are completed. For example, in the following, the word "READY" is printed each time the arm reaches the ready position, pointing straight up. If the :wait-for-move operation is omitted, then the loop will execute several times before the arm completes one cycle, and "READY" will print repeatedly, whether or not the arm is actually in the ready position.

```
(do () (nil)
  (send *puma* :ready)
  (send *puma* :wait-for-move)
  (print 'ready)
  (send *puma* :set-joint-angles '(0 0 0 0 0 0)))
READY
READY
READY
READY
[Abort]
```

Note: If you try the above example, omitting the :wait-for-move operation, you may want to recall the :abort-move operation, which can be used to stop the arm, making it forget about the 20 zillion motion commands it just received. Simply execute (send *puma* :abort-move).

2.3.3 Setting the Arm Speed

It is possible to change the speed at which the arm moves. This is done with the :speed operation.

- *Evaluate* (send *puma* :speed *speed-factor*).

The *speed-factor* is a number between 0.0 and 1.0, indicating a fractional portion of the arm's maximum speed.

The new speed takes effect immediately when the next segment begins to be servoed, rather than when the next segment to be sent to the PDP–11 is started. In other words, the new speed affects motion commands already sent to the PDP–11.

```
(send *puma* :speed 0.5)
T

(send *puma* :set-joint-angles '(0 -180 180 0 0 0))
T        ; The arm moves more slowly than before.

(progn
   (send *puma* :speed 0.5)
   (send *puma* :ready)
   (process-sleep 10.)
   (send *puma* :set-joint-angles '(0 -180 180 0 0 0))
   (send *puma* :speed 1.0))
T
```

In the latter example, we slow the arm down, send it to the ready position, then back to (0 -180 180 0 0 0), then set the speed back to 1.0. We notice that the new speed factor is used for the second segment, even though we didn't send it until after the :set-joint-angles operation. The process-sleep is there to make sure the PDP–11 starts the motion to ready before we send the speed factor. *Code must not depend on this behavior.* The way speed setting works will probably change in the future.

2.3.4 Cartesian positioning

Setting the joints of the Puma to specific angles is useful, but frequently we prefer specifying Cartesian positions relative to the "world", and offsets in x, y, and z. This is accomplished in the Puma system by the :move operation, and homogeneous transforms. (For more information about transforms, see the tutorial section Section 2.2.2, page 11, and Section 4, page 22 in the reference section.)

- *Evaluate* (send *puma* :move *transform*).

```
;;; Move the arm someplace handy.
(send *puma* :set-joint-angles '(0 -180 180 0 0 0))
T

;;; Wait for it to get there.
(send *puma* :wait-for-move)
T

;;; Record where the arm currently is.
(setq here-1 (send *puma* :here))
#<Transform Here (#46) 27472576>

;;; See what the joint angles are.
;;; (c.f. the discussion below)
(listarray (send *puma* :joint-angles))
(-0.00574977 -180.00377 179.99196 -0.0046920776 -0.0049400185 0.0008425438)

;;; Move the arm away.
(send *puma* :ready)
T

;;; Move it back.
(send *puma* :move here-1)
22.

;;; See what the joint angles are now.
;;; (c.f. the discussion below)
(listarray (send *puma* :joint-angles))
(-0.00574977 -92.47923 5.384094 -0.0046920776 87.071495 -0.023272514)
```

Depending on the configurations, the arm may not choose the same joint angles to get to the same Cartesian position. The configuration chosen is the one which causes the least movement of joints 1, 2, and 4. Thus, it is sometimes difficult to control the configuration of the arm. More control will be provided in a future release.

There are several operations available on transforms:

```
(setq here-2 (manip:transform-copy here-1))
#<Transform 27474275>

;;; Move the transform over a few millimeters.
(manip:transform-shift here-2 300 0 0)
433.07184

;;; Make the arm go there.
(send *puma* :move here-2)
23.

(manip:transform-euler-angles here-2)
-90.00942
-89.98749
179.98564

(manip:transform-set-euler-angles here-2 -90 -50 180)
#<Transform 27474275>

(send *puma* :move here-2)
24.
```

2.3.5 Cartesian "Straight line" motion

The :move operation introduced in the last section simply instructs the arm to move to the specified Cartesian position, but the arm will calculate a straight line in *joint space* to get there. To force the arm to move in a straight line in Cartesian space, use the :move-straight operation. This operation works just like :move, and, in fact, merely sends :move operations at several intermediate points along a straight line between where the arm now is, and the specified destination.

Should the arm not be where the straight line move is expected to start, (say, the arm is already executing some segments), then an optional keyword argument, :start *transform*, may be supplied. Also, the number of intermediate points calculated may also be supplied with the keyword :npoints.

```
(send *puma* :move-straight here-1)
45.

(progn
   (send *puma* :move here-2)
   (send *puma* :move-straight here-1 (:start here-2)))
67.

(progn
  (send *puma* :move here-2)
  (send *puma* :move-straight here-1 :start here-2 :npoints 2))
71.
```

2.3.6 Shutting the arm down

Once you are finished with the arm, you must gracefully shut the system down. The connection between the Lisp Machine and the PDP-11 must be closed, and the power to the controllers and PDP-11's should be turned off.

- *Close the connection between the Lisp Machine and the Puma PDP-11:*
  Evaluate (send *puma* :disconnect).

- *Turn off power to the Unimation controller and both PDP-11's.* Important: Make sure the arm power is off before turning off the Unimation controller. *If the arm power is still on, (the red light above "ARM POWER ON" is lit), the arm may jerk, causing damage to the arm or to someone standing in its path.*

It is not imperative to turn off the PDP-11's and the controller. For example, if you are only going to be leaving for a short time, they may be left on, and only the connection be closed. It is wise, however, to make sure either that the arm power is off (by pressing either of the panic buttons), or that the connection is closed (automatically shutting off arm power), if you will be leaving the room for any length of time.

```
(send *puma* :disconnect)
Do you really want to disconnect from the Puma? (Y or N) Yes
"Disconnected."
```

3 Advanced Features

In this section, we examine compliant motion and guarded motion.

3.1 The Compliance Equation

Puma's response to the forces in its environment is determined by the compliance equation and by the chosen location of its center of compliance. The equation relates the following terms:

- The force the Puma is commanded to exert
- The force it senses
- The position it is commanded to achieve
- The position it actually reaches
- The velocity with which it is commanded to move
- Its actual velocity

The degree to which each of these terms contributes to the actual motion of the arm may be determined by the programmer.

The compliance equation implemented by Cougar is:

$$\Delta\dot{\theta} = B_j^{-1} J^T [(F_C - F_S) + KJ(\theta_C - \theta_S) + B_{Car} J(\dot{\theta}_C - \dot{\theta}_S)]$$

Where we have:

$\Delta\dot{\theta}$	The velocities to be added to the joint velocities of a non-compliant trajectory, a 6 vector.
B_j^{-1}	Inverse of the joint damping matrix, a 6 element diagonal matrix.
J^T	Jacobian transpose in the frame of the compliance center, a 6×6 matrix.
F_C	The commanded cartesian force-torque, a 6 vector of forces in the compliance center frame.
F_C	The transformed force-torque as measured by the wrist sensor.
K	The spring constant matrix, a 6 element diagonal matrix.
J	The Jacobian.
θ_C	The vector of commanded joint positions 1 through 6.
θ_S	The vector of sensed joint positions.
B_{Car}	The cartesian damping matrix, a 6×6 matrix.
$\dot{\theta}_C$	The vector of commanded joint velocities.
$\dot{\theta}_S$	The vector of sensed joint velocities.

A nominal trajectory for the arm is calculated by Puma, when compliance is active Cougar modifies that trajectory according to the above equation. Each servo cycle Cougar measures the forces on the wrist and receives from Puma the sensed joint angles, θ_S, the commanded angles, θ_C, and the difference between the commanded and sensed joint velocities, $(\dot{\theta}_C - \dot{\theta}_S)$. After computing the compliance equation Cougar returns a compliance correction velocity, $\Delta\dot{\theta}$, which Puma adds to the velocity it calculates for a non-compliant trajectory. This sum specifies the ideal trajectory of the arm.

Overall the form of the equation is that of a joint damper: $\Delta\dot{\theta} = B_j^{-1} Force$, velocity is proportional to force. The B_j factor largely determines the stability of the system, it limits the response of the arm

to applied forces. The bracketed sum in the compliance equation is converted to a joint space force by multiplication by the Jacobian transpose, J^T.

Each term within the brackets has the dimensions of a Cartesian force. The the first term, $(F_C - F_S)$, allows the user to choose the static force exerted by the arm. If the other terms were set to 0 Cougar would compute a compliance correction velocity which would cause the arm to move in such a way as to reduce the difference between the commanded and sensed forces to 0.

The second term, $KJ(\theta_C - \theta_S)$, manufactures a force proportional to the difference between the commanded and sensed position. The diagonal matrix K thus lets the user pick the stiffness, that is the restoring force due to a displacement from the commanded position.

The final term, $B_{Car}J(\dot{\theta}_C - \dot{\theta}_S)$, generates a force which damps out differences between the commanded and sensed Cartesian velocities. Because B_{Car} is a 6×6 matrix the user may relate any component of the commanded velocity to any component of the sensed velocity.

3.2 Compliant Motion

When the arm moves compliantly, in accordance the above equation, there is a question as to when the motion segment is completed. The Puma resolves that question by pretending that the arm is not complying—that is, that the motion will be stopped when the nominal trajectory would stop, regardless of where the arm actually is.

Setting the Parameters

For compliant motion, we must specify the matrices used in the compliance calculation. As described in Section 6.5, page 35, the matrices which may be specified are the desired force, spring constant diagonal, cartesian damping matrix, joint damping inverse diagonal matrix, and the frame vector. Each of these may be specified using the `:comply-parameters` operation:

- *Evaluate* (`send` *puma* `:comply-parameters` *matrix-type matrix*).

The *matrix-type* should be one of the keywords `:force`, `:spring-constant`, `:cartesian-damping`, `:joint-damping`, `:frame-vector`, or `:effective-weight`. The *matrix* should be an array or list with the appropriate number of elements. That is, six values for the desired force, spring constant, and joint damping inverse matrices, thirty-six values for the cartesian damping matrix, four for the effective weight, and three for the frame vector.

An operation which allows the user to specify all of the matrices at once is `:initialize-compliance`. Any of the matrices may be specified, and those which are not are set to all zeroes, except for the joint damping matrix, which is set to the values shown in the example below. See the reference section for more details.

```
(send *puma* :comply-parameters :force '(0 0 0 0 0 0))
1.

;;; :stiffness is a synonym for :spring-constant
(send *puma* :comply-parameters :stiffness '(0 0 0 0 0 0))
2.

(send *puma* :comply-parameters :joint-damping
      '(0.0000065 0.000006 0.0000245 0.000195 0.00012 0.001))
3.
```

```
(setq b (make-array '(6 6) :initial-value 0.0))
#<ART-Q-6.-6. 32377651>

(send *puma* :comply-parameters :cartesian-damping b)
4.
```

Normally, Cougar bootstraps with reasonable values for these parameters (mostly zero), but it is a good idea to set them anyway before doing compliant motion. The values shown above for the joint damping matrix are stored in the variable puma:good-values-for-joint-damping, for easy reference.

Calibrating the Strain Gauges

Also, before complying, the strain gauges in the wrist must be calibrated. To do this, the arm must be moved so that the hand can be made to point straight up and then straight down (to compensate for gravity). An operation, :calibrate-wrist-gauges, is defined to perform this calibration:

```
(send *puma* :calibrate-wrist-gauges)
May I move the arm to [0, -90, 180, 0, -90, 0] (wrist pointing up)? (Y or N) Yes.
20 gauge readings pointing up:  -10, -20, 30, -40, 440, 241, 42, 1
May I move the arm to [0, -90, 180, 0, +90, 0] (wrist pointing down)? (Y or N) Yes.
20 gauge readings pointing down:  -53, -20, -3, -43, 400, 243, -7, 5
Average readings:  177747, 177760, 12, 177737, 420, 242, 15, 3
Send them to the wrist? (Y or N) Yes.
T
```

Compliant Motion

Now, we are ready to comply. The same operations that we learned in the preceeding tutorial section are used for compliant motion. The only difference is that the keyword argument :compliantly t is added to the call:

```
(send *puma* :set-joint-angles '(0 0 0 0 0 0) :compliantly t)
T

(listarray (send *puma* :joint-angles))
(2.5643973 7.963188 21.224022 13.427372 30.800848 1.649867)
```

We can see that the arm didn't exactly get to where we specified. (I was pushing on the hand quite hard.) This works for all the motion operations, :set-joint-angles, :move, and :move-straight.

Summary

To summarize the steps needed for compliant motion:

- *Calibrate the wrist strain gauges with the* :calibrate-wrist-gauges *operation.*
- *Send the desired parameters to the wrist PDP-11, using* :comply-parameters, *or* :initialize-compliance.
- *Comply. Use the* :compliantly t *argument to the motion operations.*

3.3 Guarded Motions

The user may desire certain force events to halt arm motion and/or generate a report of their occurrence. Guarded motions make this possible. Three parameters are necessary to specify such a motion:

1. The direction, in hand coordinates, in which the force-torque is monitored.
2. The force-torque magnitude at which to take action.
3. Whether to halt the arm or return a message when the anticipated event occurs.

Cougar implements guarded motions by recording the three quantities above for each guard as it is established. Every servo cycle afterwards Cougar checks "untripped" guards by forming the dot product of each guard's direction vector and the current force-torque reading from the wrist and comparing it to the recorded magnitude. The guard "trips" if it ever happens that:

$$Magnitude \geq Direction\ Vector \cdot ForceTorque\ Vector$$

When this does occurs one of two messages is sent to Puma, either a 'stop-move' or a 'continue-after-trip.' The guard then becomes inactive and is not checked again unless reset by the programmer. At most 16 guards may be active at any one time.

The `manip:with-guard-set` special form makes it easy to set guards around a section of code. The reference section contains a detailed description of guards and `manip:with-guard-set` (see Section 8, page 45). Here we will just give a simple example of the use of the special form.

```
;;; This guard will detect a force in the z direction of 20 ounces.
(manip:with-guard-set (*puma* :vector '(0 0 1 0 0 0)
                              :tolerance 20.0)
   (send *puma* :set-joint-angles '(0 0 0 0 0 0))
   (send *puma* :wait-for-move))
NIL
```

Part II
Reference Material

4 Transformations

The Puma system uses 4 × 4 homogeneous transformations to represent positions and orientations of the hand in Cartesian coordinates. This document does not discuss the meaning and use of transformations; that information is easily found elsewhere (for example, in *Robot Manipulators*, by Richard P. Paul, *op. cit.* on page 2.2.2). This section discusses their implementation in the Puma system and operations which are available for manipulating them. (Note: here and elsewhere in this document the words "transformation" and "transform" are used interchangeably.)

Transforms are represented in the Lisp world by a named structure of type `transform`. This implementation was chosen to provide for generality, should other representations for Cartesian position and orientation be constructed. Objects which are expected to behave like transforms should be implemented as named structures which provide at least the functionality to be described presently. Programs may be written using the functions described in this section for manipulating transforms and should then work on any object which is defined to behave like a transform.

By special dispensation, simple 4 × 4 `art-q` arrays are accepted by all the transform functions. Flavor instances are also accepted, and should be defined to accept messages corresponding to all the operations described in section 4.2.

4.1 Transforms functions

The functions described in this section are available for creating, manipulating, and examining transforms or objects which behave as transforms.

The orientation of the hand may be specified by a set of three Euler angles. The tranform operation `manip:transform-euler-angles` may be used to read the Euler angles and `manip:transform-set-euler-angles` may be used to change the Euler angles associated with the transform's rotation matrix.

Figure 3 demonstrates Puma's definition of the Euler angles. Choose a coordinate system oriented relative to the hand as shown—with the fingers in a horizontal plane and pointing in the negative y direction. The first rotation, O, is clockwise about the z axis. It generates two new axes x' and y'. The illustration follows the x-y plane as it is transformed by the rotations.

The next rotation, A, counterclockwise about the x' axis, creates two more new axes y'' and z'. Lastly, a *clockwise* rotation of T is made about the y'' axis. The crosshatched area shows the final orientation of what was originally the x-y plane. The axes are x'', y'', and z''.

The three angles thus defined can also be thought of as a *yaw*, *pitch*, and roll of the hand.

manip:make-transform &optional *initial-values name* *Function*

Returns a `transform`. If *initial-values* is supplied, it should be an object suitable for the second argument to the `fillarray` function; it is used to initialize the contents of the transform. Any unspecified elements of the transform are initialized to zero. (To get a null transformation, use `(manip:null-transform)` (q.v.).

The transform is "named" if the *name* argument is supplied. No use is made of this name, except for printing the transform, where it helps to identify it. (All transforms extracted from responses from the Puma PDP-11 are named for the request which elicited the response.)

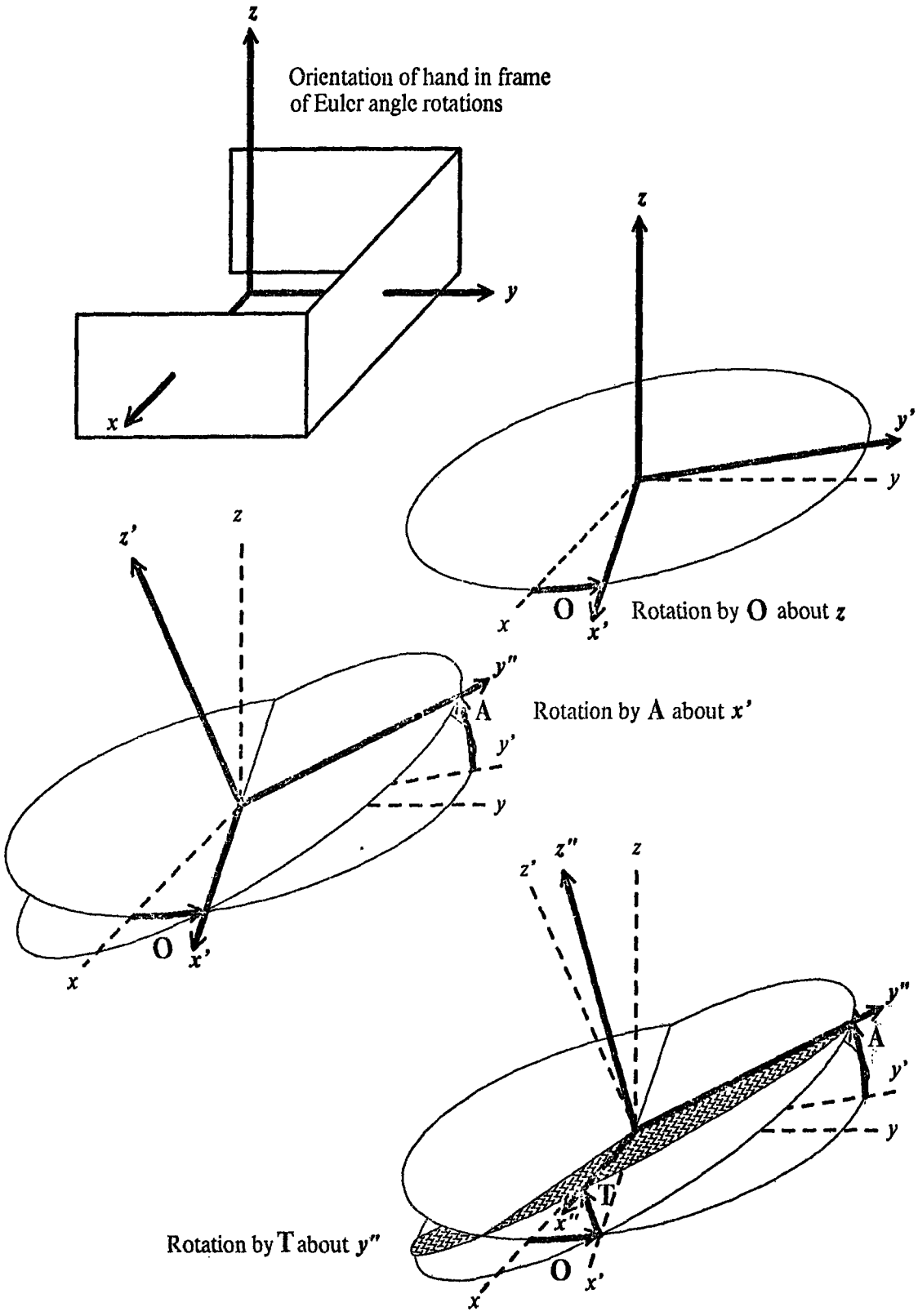

Figure 3: The Puma's definition of the Euler angles.

manip:transform-null-transform *Function*

Returns the null transformation. This is the 4×4 identity matrix, and, in fact, is really a copy of the transform in the variable `the-4x4-identity-matrix`.

manip:transform-describe *transform* &optional *stream* *Function*

Prints a description of *transform* on *stream*. The description includes a display of the entries of the transform.

manip:transform-array *transform* *Function*

All transforms include as their means of representing a transformation between coordinate systems a 4×4 matrix of numbers. This function will return this array, so that the user program may access the elements of this array with `aref` and `aset`. The user should be aware, however, that some implementations of transforms may not use this array as the interesting part of their representation, and that changing elements of this array may be ineffectual, and that the array may not be updated should other alterations to the transform be performed.

manip:transform-copy *transform* *Function*

Returns a copy of *transform*. The copy is identical in all respects, with the exception that the new transform will not have a name, even though the original may have. This function is quite useful, as most of the other functions which manipulate transforms change them.

manip:make-new-transform *old-transform array* *Function*

This function returns a new transform of the same type as *old-transform* (i.e. array, named structure, instance, etc.), with a transformation matrix corresponding to *array*. Using this, for instance, one could define `manip:transform-copy` as follows (this is not how it's defined, however):

```
(defun manip:transform-copy (transform)
  (manip:make-new-transform transform
    (manip:transform-array transform)))
```

manip:transform-position *transform* *Function*

Returns three values, the x, y, and $z$coordinates represented by the transform.

manip:transform-set-position *transform x y z* *Function*

Alters the transform to correspond to the coordinates specify by x, y, and z.

manip:transform-euler-angles *transform* *Function*

Returns three values, the Euler angles O, A, and T.

manip:transform-set-euler-angles *transform O A T* *Function*

Sets the rotation matrix of the transform to correspond to the Euler angles O, A, and T.

manip:transform-compose *transform* &rest *transforms* *Function*

This function returns a new transform which is the result of composing all the transforms in the order specified. This is essentially performed by multiplying all the arrays for all the transforms, then creating a new transform with the resulting array. The new transform will be of the same type as *transform*.

manip:transform-invert *transform* *Function*

Returns the inverse transformation to *transform*. The new transform is of the same type as *transform*.

The following functions do not take transforms, *per se*, as arguments, but instead take transformation matrices. They are defined this way for effeciency, as they are most useful in situations where much mathematical computation is being performed. The user may find them useful. These functions refer to the column vectors of the transform matrix, designated in the literature as **n**, **o**, **a**, and **p**.

manip:xform-dot *array vec1 vec2* *Function*
Returns the dot product of the two column vectors. The vectors are specified by numbers, where **n**, **o**, **a**, and **p** are represented by 0, 1, 2, and 3, respectively.

manip:xform-n *array index* *Function*
Returns the *index*th element of the **n** vector of the transform.

manip:xform-o *array index* *Function*
Returns the *index*th element of the **o** vector of the transform.

manip:xform-a *array index* *Function*
Returns the *index*th element of the **a** vector of the transform.

manip:xform-p *array index* *Function*
Returns the *index*th element of the **p** vector of the transform.

4.2 Details of the Transform Structure Implementation

As described above, transformations are implemented by named structures, and that the functions listed in the previous section will accept any named structure providing a certain set of operations. This section will list the required operations. Most of these operations are just the implementation of the above described functions, though a few exist for special purposes. The operations are implemented either as methods, for instances, or as a "named-structure-invoke" function for a named structure. Refer to the Lisp Machine documentation on `defstruct` for details on the latter.

This section should supply sufficient information for a user to create his or her own representation for Cartesian positions. This is relevant, because the protocol for the Puma PDP–11 requires transforms. Whenever a named structure, two dimensional array, or instance is passed to the Puma, it is expected to yield a transform in some form or another (see below). Also it allows programs to be written to accept any object which will behave as a transform without depending on a particular representation.

Suggestions and/or implementations of alternate representations are welcomed by the author. This list of required operations will be expanded as other representations are adopted.

`:transform` Implements `manip:transform-array`.

`:copy` Implements `manip:transform-copy`.

`:make-new-transform` Implements `manip:make-new-transform`.

`:euler-angles` Implements `manip:transform-euler-angles`.

`:set-euler-angles` Implements `manip:transform-set-euler-angles`.

`:position` Implements `manip:transform-position`.

`:set-position` Implements `manip:transform-set-position`.

`:shift` Implements `manip:transform-shift`.

`:describe` Implements `manip:transform-describe`.

`:transform-print` Implements `manip:transform-print`.

`:transform-compose` Implements `manip:transforms-compose`. This operation however, only accepts two transforms. The first is of the type handling the operation, but the second argument may be of any type. The implementor should be aware that `manip:transform-compose` may assume that composition of transforms of any type is associative, though not commutative.

`:fast-transform-compose` This is optional, and may assume that both transforms are of the same type.

`:transform-invert` Implements `manip:transform-invert`.

`:fill-packet` Takes as arguments the named structure, an `art-16b` array (the packet array), and an index into that array. This operation should store into the array the twelve interesting elements of the transform array. These are the first three elements of each column, in column major order. The each element must be stored as two consecutive 16-bit numbers, representing the two halves of the 32-bit single precision PDP–11 floating point number. The PDP–11 uses these values directly. This operation must return the array index of the next free element in the packet array, which will be the starting index plus 24.

`:dump-packet` This operation must alter the transform to reflect the transform array stored in the packet array. Arguments and data format are as described in the `:fill-packet` operation.

The following operations are not required, but may be supplied if appropriate.

`:name` Returns the "name" of the transform. The name is mostly useful for displaying to the user.

`:set-name` Changes the "name" of the transformation.

`:print-self` The printed representation of the object. See the Lisp Machine documentation.

5 Representation and Protocol

This section contains a brief discussion of the operation of the background process which maintains the connection with the PDP–11. The user may be interested in this, as it help explain the behavior of some of the operations.

The Lisp Puma system is controlled primarily by a background process which accepts requests from other, user, processes and forwards them on to the Puma PDP–11. When the Puma PDP–11 has satisfied the request, it sends a response, possibly containing data, back to the Lisp machine. The background process accepts the response, decides which request it belongs to, extracts any data from the response packet, possibly informs the user process which made the request that the response has arrived, and handles any error conditions which are reported by the PDP–11, or detected by the background process itself.

The communication between the background process and the user processes is through the puma flavor instance, and through the request objects (see Section 5.3, below). There is one exception to this; see Section 7.3, page 42. The puma flavor also controls the operation of the background process, including how it sends off requests over the network and how it interprets responses. This is how the background process may be tailored to a specific manipulator. See the separate document, "An Implementor's guide to the Lisp Machine Puma System."

The remainder of this section describes in a little more detail the operation of the user request and requests in progress queues, motion requests, and the request objects themselves.

5.1 Queues

There are three separate queues maintained by the background process. A "user request queue," a "requests in progress queue," and a "response queue." The user request queue is for requests which have been accepted by the background process, but which have not been sent to the PDP–11. The requests in progress queue contains requests which have been sent to the PDP–11, but which have not received responses. The response queue contains all the responses which the PDP–11 has sent, but which have not yet been processed.

When a user process requests an operation of the Puma, the request object is added to the user request queue. The process is then free to go on doing any processing it desires, safe in the knowledge that the request will get to the Puma PDP-11; it need not wait for a response, or do anything else with respect to that request—not now, not ever. In fact, most of the defined user operations which do not expect data in return do just that—they queue the request, then return to the program. Of course, there will be some delay between the time the operation returns and the time the Puma PDP-11 actually gets around to processing the request. (See the section on requests for the functions a program may call to wait for the response and get data from a request.)

The background process takes request objects off the user request queue and sends a network packet to the PDP-11. The request is then added to the requests in progress queue. The PDP-11 will handle the request and send back a response. By the definition of the protocol, the PDP-11 will always send a reply or an error packet in response to any request, and will always handle requests in the order they are sent. Replies always are sent in the same order as the requests they are in response to. Further, each response contains a sequence number indicating to which request it corresponds. The background process considers it an error if it receives a response with a different sequence number than the request it believes is next in line for a response.

When the background process receives a response or an error packet it matches it with the request at the head of the requests in progress queue (signalling an error if it does not belong to that request). If the request is expecting any data in the response packet, that data is extracted from the packet. (Note that if the request does not expect data that is actually in the packet, that data is ignored.) A flag is set in the request object to notify any process that is waiting, or will wait, for the response to the request, that the response has, indeed, arrived. If the "response" was really an error packet, then that fact is communicated to the user process(es) through the request object.

There are a few ramifications of this procedure of which the user should be aware. For smooth and efficient operation of the protocol, the background process will send several (currently two or three) requests off to the PDP-11 without waiting for the responses to the first request to arrive at the Lisp Machine. Also, there may be several responses waiting in the response queue, and more requests may be sent. Thus, at any given time, there may be several requests and responses "floating in the network." That is, requests sent by the Lisp Machine (or PDP-11, for responses) and not yet started to be handled by the PDP-11 (or Lisp Machine, for responses). Under certain circumstances, the Puma PDP-11 may find itself in a condition where no further requests may be accepted until the condition is cleared up. It can inform the Lisp Machine, which can flush any pending queue entries, but neither the Lisp Machine, nor the Puma PDP-11 can do anything about those packets backed up in the network. The background process attempts to minimize the effects of this by handling any received responses or error packets before sending any request packets, and by checking for responses after sending each request, but the situation can still occur. The problem arises so infrequently, however, that it is not reasonable to sacrifice the efficiency of the present procedure for the slight chance that it will be a problem.

In summary, the user processes communicates its desire to make a request of the Puma by adding a request object to the user request queue maintained in the puma flavor instance. Then, independent of the user process, the background process sends the request to the PDP-11, fields the response from the PDP-11, and communicates the response data and the fact that the response has arrived back to the user process through the request object.

5.2 Motion Requests

Requests for arm motion are handled slightly specially. The discussion above is still exactly correct, but the handling on the PDP-11, and its relationship to the above described protocol can be confusing.

In order to be able to plan a smooth trajectory with several set points, the PDP-11 is programmed to be able to accept several move-type requests and "handle" them while the arm is being servoed along a

trajectory specified by earlier motion requests. The PDP–11 maintains an internal queue of motion segments, added to by each move request, and processed in sequence.

The handling of motion requests is as follows: When the Puma PDP–11 receives such a request, the joint angles to which to move are added to the move segment queue. The response for that request is then returned to the Lisp Machine immediately, without waiting for the motion to either start or stop. This means that subsequent requests, either for motion or for other things, may be handled and answered while the motion is in progress, or even before the motion starts. However, this also means that the user process has no way of knowing when the motion that it requested will either start or stop.

The `:wait-for-move` operation described in Section 6.3, page 34 does allow the user process to wait until a given trajectory completes, however it does have a serious problem. The Puma PDP–11, in handling the request, itself waits until the arm is no longer moving. Thus, no other requests are recognized, accepted, or handled while the arm is being waited for. This includes, unfortunately, such operations as `:abort-move` and `:kill-power`. It is not possible to change this behavior within the bounds of the protocol, nor is a special case check for `:abort-move` possible. In the future, better mechanisms for these operations may be developed.

5.3 Requests

Request objects contain all the information the background process needs to construct the network packet to send to the Puma PDP–11, to retrieve data out of the packet, and to notify the user process(es) of the arrival of the response or error packet. This section details just what that information is, and what functions are available for manipulating requests.

The request object is a structure containing the following slots:

- Slots pertaining to the formation and sending of the network packet:

  manip:request-code Contains the numeric code indicating what the request is—describe, move, joint angles, etc.

  manip:request-data Contains whatever data needs to be sent over to the Puma PDP–11 for the proper handling of the request. `NIL`, if no data is required.

  manip:request-immediate? A flag, saying whether the request should be sent ahead of any other requests on the user request queue.

  manip:request-packet An internal slot.

- Slots pertaining to the handling and interpretation of responses:

  manip:request-response-type A keyword, `:single` or `:multiple`, indicating that the response to this request is contained in a single packet or a sequence of packets.

  manip:request-response-data-extractor Contains a function which will extract the data from the response packet and store it in the `manip:request-data-drop-off` slot. `NIL` if no data is expected, or if the data is to be ignored.

  manip:request-data-ready? A flag, set when the response has arrived and all the data (if any) has been extracted from it. If the response was an error packet, this slot will contain a condition object identifying the error. It is not good to explicitly test this flag to see if a response has arrived. The function `manip:wait-for-response` (see below) performs this function, but also does some useful error checking.

  manip:request-data-drop-off This is where any data from the response packet is stored. If there was any data, and `manip:request-response-data-extractor` was non-`NIL`, then this slot will be set to the appropriate data object, and then the `manip:response-data-ready?` flag will be set, in that order.

manip:request-interrupt Contains code to be executed when the response is handled. This code will be executed after `manip:request-data-ready?` is set. See the description of the `:make-request` operation for what may go in this slot.

- Slots pertaining to notification of processes and error handling:

manip:request-packet-number An internal slot. Used to match responses with requests.

manip:request-requesting-process This points to the process which created the request. For special applications, this may not necessarily be the same process which queues the process. See Error Signalling, Section 7.3, page 42, and the function `wait-for-response` for the use of this slot.

manip:request-error-signal May contain a process, or `NIL`. See Error Signalling, Section 7.3, page 42.

manip:request-process-waiting Contains the process which is waiting for the response to this request, or `NIL`.

It is rarely necessary to access the request object directly. Special applications, however may wish to do so. The user should be aware that the background process expects most of these slots to be in a special format. Refer to the "Implimentor's Guide" for details.

Requests may be constructed using the `:make-request` operation of the puma flavor, or by calling `make-manipulator-request` directly (see the code, or the "Implementor's Guide"). The following functions may be useful for manipulating requests:

manip:wait-for-response *request* &optional *(who-state "Request")*
&key *not-me-ok (error-signal t)* *Function*
Waits until the response is received for *request*. If the request resulted in an error, then an error condition is signalled. This feature is inhibited, however, if *error-signal* is `NIL`.

It is considered an error to wait on a request which was created by a different process (see the request slot `manip:request-requesting-process`, above), unless the argument *not-me-ok* is supplied non-`NIL`.

manip:request-queue-enqueue *request queue* *Function*
Adds *request* to the end of *queue*. If (`manip:request-immediate?`*request*) is non-`NIL`, then the request is added to the head of the queue.

manip:request-queue-dequeue *queue* *Function*
Returns the request at the head of the queue, and removes it from the queue. Signals an error if the queue is empty.

manip:request-queue-peek *queue* *Function*
Returns the request at the head of the queue, but does not remove it from the queue. Signals an error if the queue is empty.

manip:request-queue-empty *queue* *Macro*
Returns `T` if the queue is empty, `NIL` if there are packets in it.

manip:clear-request-queue *queue* *Function*
Forces the queue to be empty. Does nothing to or with the requests which may have been in the queue.

manip:make-request-queue *Macro*
Makes a request queue. This is actually a structure constructor, and takes initializations for the slots `manipulator` and `manip:name`. See the Lisp Machine documentation for `defstruct`.

6 Puma Operations

This section presents a complete list of the user operation which are handled by the Puma. They are listed by category of operation. Following is a summary of the categories and operations. The subsequent sections contain the detailed descriptions of each operation.

- Initialization
  :calibrate
  :calibrate-wrist-gauges
  :disconnect
  :initialize
  :initialize-manipulator-description
- Informative
  :degrees-of-freedom
  :describe-manipulator
  :describe-version
  :handle-operation?
  :here
  :joint-angles
  :manipulator-type
  :operations
  :read-forces
  :read-gauges
  :unique-name
- Motion
  :abort-move
  :enable-arm-power
  :free-joint
  :move
  :move-straight
  :ready
  :reset-abort-condition
  :set-joint-angles
  :speed
  :wait-for-move
- Hand
  :close-hand
  :hand
  :open-hand
- Compliance
  :comply
  :comply-end
  :comply-parameters
- Guards
  :clear-guard
  :set-guard
- Miscellaneous
  :connection-open?
  :deposit
  :examine
  :flush-queues
  :get-parameter
  :notify-trivial-errors?
  :send-self-and-wait-until-done
  :set-notify-trivial-errors?
  :set-parameter
  :set-trivial-error-codes
  :solve-backwards
  :solve-forwards
  :trivial-error-codes
  :wait-until-no-requests-pending
  :wrist-deposit
  :wrist-examine
  :write-gauge-offset-vector
- Requests
  :make-request
  :request
  :send-request

6.1 Initialization

The following operations are used to set up the Puma system for operation. The `:initialize` operation is required, as it creates the network connection to the Puma PDP–11, and must be performed before any other requests may be processed. The `:calibrate` operation is important, to ensure that the arm is positioned repeatably between sessions.

:initialize &key *quietly? notify-errors?* *Message*
Creates a connection to the Puma server and initializes the arm. If *quietly?* is specified, then the user is not asked for confirmation.

:calibrate &key *(wait-p nil)* *Message*
Causes the PDP–11 to calibrate the joint angle encoders. Each joint is moved slowly a small amount until a zero index on the encoder is found. A potentiometer in the joint is read, and the encoder reading corresponding to that potentiometer value is then stored in the controller as the current encoder count. If the *wait-p* argument is supplied non-nil, then the operation will wait for the PDP–11 to respond before returning. *This operation causes the arm to move.*

:disconnect &key *(confirm-p t) (mouse-confirm-p nil) (force nil) (wait t)* *Message*
Closes the connection to the Puma server. *Wait* says to wait until all the pending requests have been serviced. *Force* indicates that any pending requests should be flushed before disconnecting. *Force* is more powerful than *wait*. The confirm arguments indicate whether the user should be queried for confirmation.

If any use of wrist force measurements is to be made, such as compliant or guarded motion, the `:calibrate-wrist-gauges` message should be sent. This will ensure reasonable force readings.

:calibrate-wrist-gauges &key *(readings 20.) (verbose t) (query? t)* *Message*
This function will move the wrist (and, hence, the arm) such that the hand will be pointing upward, take several strain gauge readings, then take several gauge readings with the hand pointing downward. It will average all the readings, then write the subsequent average back to the wrist PDP–11. *Readings* specifies the number of readings to take with the wrist pointed in each direction. The *verbose* flag controls whether the average gauge readings are displayed; the *query?* flag controls whether the user is asked for confirmation before moving the arm each time, and before sending the data back to the wrist.

The following operation is not normally necessary, but is documented here for completeness.

:initialize-manipulator-description *Message*
The puma flavor instance stores a cache of the information retrieved by a `:describe-manipulator` request. This operation causes the Lisp puma object to refresh that cache from the data provided by the PDP–11. This is automatically done when the connection is initialized, and normally need not be done manually.

6.2 Informative

The Puma can supply the user with several types of information, including what it is, what it can do, and where it is.

The following operations allow the user to examine various aspects of the Puma. Although not generally interesting for just using the Puma, these operations would be useful for a general program that could work with several different manipulators.

:degrees-of-freedom *Message*

Returns the number of degrees of freedom of the manipulator. Always returns 6.

:describe-manipulator *Message*

Sends a DESCRIBE packet to the manipulator and returns the list generated. This is a list of alternating keywords and values describing various parameters of the arm. The defined keywords and their values are:

`:brand-name` A string giving the brand name for the type of manipulator. For example, `"Unimation Puma 600"`.

`:unique-name` A string giving the unique name of the arm, such as `"Puma #1"`.

`:software-version` A list of alternating keywords and version numbers. Each keyword corresponds to a module of the PDP-11 software. The keywords `:day`, `:month`, and `:year` are also provided, and specify the date on which the Puma 11 software was assembled.

`:protocol-version` A number, indicating what version of the communication protocol the PDP-11 is running.

`:degrees-of-freedom` 6 for the Puma.

`:operations` A list of the operation codes handled by the server. These may be used to check to make sure the server actually handles a request.

:describe-version &optional *(stream standard-output)* *Message*

Prints on *stream* the modules and versions of the software running in the PDP-11. This is just a formatted version of the `:software-version` information from the `:describe-manipulator` operation.

:handle-operation? *operation* *Message*

Returns T iff the specified *operation* is handled by the Puma server. *Operation* may be either a numeric operation code, or a symbol representing such a code. This should not be confused with the Lisp system provided `:operation-handled-p`, which refers only to operations on the flavor instance. `:Handle-operation` refers to manipulator protocol operations.

This message only checks the list of supported operations which is locally cached in the flavor instance. See `:initialize-manipulator-description`, page 31.

:manipulator-type *Message*

This operation always returns `:puma` for the Puma manipulator. Presumably, it will return something different for other manipulators.

:operations *Message*

Returns a list of the operation codes handled by the Puma server. This will return the same information as `(cadr (memq :operations (send` *puma* `:describe-manipulator)))`, but does not go to the PDP-11 for the information. (See `:initialize-manipulator-description`, page 31.)

:unique-name *Message*

Returns the "unique name" of the manipulator. See `:describe-manipulator`, page 32.

There are two different ways the Puma can report its current position. Either as a vector of joint angles, or as a homogeneous transformation representing the position and orientation of the hand in Cartesian coordinates. The following two operations supply this information.

:here *Message*

Get a transformation corresponding to the state of the arm. Returns a homogeneous transformation

representing the displacement and orientation of the Puma's hand with respect to the base of the manipulator. See sections 2.2.2 (page 11) and 4 (page 22).

:joint-angles *Message*

Returns an array of six elements, floating point numbers, representing the six joint angles in degrees.

The following operations retrieve information from the wrist force sensor.

:read-forces *Message*

Reads the forces on the wrist. Returns an array of six elements, representing the forces along the x, y, and z axes of the hand frame, and the torques around those axes. The forces are in units of ounces, the torques in ounce-inches. The torques are measured about the center of compliance.

:read-gauges *Message*

Returns the raw strain gauge readings from the wrist. This is used mostly for debugging the wrist PDP-11, and for calibrating the force calculation. See `:calibrate-wrist-gauges`, page 31.

Various miscellaneous parameters may be examined by the `:get-parameter` operation. See Section 6.7, page 37.

6.3 Motion

This section presents the various ways to actually make the arm move. Several other motion related operations are also presented.

:move *transform &key compliantly* *Message*

The Puma 11 solves the inverse kinematics for the specified *transform* (see Section 4), and moves the arm to that point in space. If there is no solution, or the solution requires joint angles outside the joint limits, an error is signalled. If *compliantly* is specified non-nil, then compliance is used during the move.

Actually, this request only queues the point to which to be moved, then returns. Any motion segments queued earlier are finished before this one is started, with smooth transitions between segments.

In general, there are eight possible solutions to the inverse kinematics for the Puma arm. The PDP-11 chooses the solution which involves least motion of the first two joints.

:move-straight *transform &key compliantly start (npoints 20.) solve-first* *Message*

Like `:move`, but approximates a straight line in Cartesian space for the motion. The approximation is accomplished by linear interpolation between the starting and finishing transformations, with the number of interpolated points being specified by *npoints*. To specify a starting point different from the current arm position, specify *start* as the starting transformation. If *solve-first* is non-nil, then the Puma 11 is asked to solve each of the transforms for the corresponding joint angles before sending any of the joint angles to the PDP-11 for motion.

:ready *Message*

Moves the arm to the "ready" position—pointing straight up toward the ceiling. Same as `(send` *puma* `:set-joint-angles '(0 -90 90 0 0 0))`.

:set-joint-angles *destination &key compliantly wait-p* *Message*

The arm will move to the joint angles specified by *destination*, which must be either a list or an array of six elements. As usual, *compliantly* will turn on compliance during the motion. If *wait-p* is specified, the operation will wait until the motion segment is queued in the PDP-11 before returning, and will return the segment number. Otherwise, it will return immediately.

:speed *speed-factor* *Message*

Sets the speed at which the arm will move. The *speed-factor* is a number between 0.0 and 1.0, indicating a fractional portion of the arm's maximum speed.

The new speed takes effect immediately when the next segment begins to be servoed, rather than when the next segment to be sent to the PDP–11 is started. In other words, the new speed affects motion commands already sent to the PDP–11.

:wait-for-move *Message*

This operation causes the PDP–11 to cease processing requests until all currently queued motion is completed. Note: This means that even requests such as `:abort-move` or `:examine` are not processed. This restriction will be lifted in a future release of the system.

The following operations deal with aborting motion, and recovering from such conditions.

:abort-move &key *(wait-p nil)* *Message*

Causes the currently executing trajectory to be stopped. The arm is stopped where it is, immediately, without decelerating. Any segments already queued in the PDP–11 are flushed. This operation is a no-op if the arm is not moving and no segments are queued. If the *wait-p* argument is supplied non-nil, then the operation waits until the PDP–11 responds before returning.

:enable-arm-power *Message*

Most of the time, the arm power is software-enabled. That means that when you push the black "ARM POWER ON" button, the power will go on. However, it is possible to turn off the power under program control. This is done when the PDP–11 decides that a collision has occurred, for instance. Under these circumstances, the user program must re-enable arm power after, presumably, having the problem cleared up. The `:enable-arm-power` operation performs this service.

:reset-abort-condition &key *(clear-queues t)* *Message*

This operation is used to recover from situations where the PDP–11 has decided that it was unable to completely satisfy all the queued motion requests. For instance, consider the following scenario: Several motion requests have been queued in sequence, and the arm is merrily moving along. Suddenly, the user hits the panic button, cutting off power to the arm. The PDP–11 notices this, and aborts the queued trajectory, then informs the Lisp Machine that there is no arm power and waits for further instructions. Due to the asynchronous nature of the packet communication between the two computers, the PDP–11 requires some means of knowning which subsequent requests are based on the new information that the arm has stopped moving. The `:reset-abort-condition` satisfies this requirement. When the PDP–11 receives this "request", it then may assume that any new motion requests it receives will be based on the understanding that some already requested motion was not successfully completed.

If any motion type requests are made before sending this request, the PDP–11 will return an error message, and refuse to handle the request. This includes such requests as `:comply-parameters`, and `:set-guard` and `clear-guard`.

The *clear-queues* flag indicates whether or not the user-request-queue and the requests-in-process-queue should be cleared before queueing the reset request.

Those users who wish to have their programs handle error conditions and issue this request automatically should carefully consider that requests may be queued in the network before the PDP–11 discovers and reports the abortive condition, and these may not be retracted. The PDP–11 will then issue "arm motion was aborted" error messages that the Lisp Machine, in some sense, is expecting.

It is not often necessary to make a joint go limp, but occasionally it can be useful. Especially after (accidentally, of course) jamming the arm into the table, or if a joint goes out of range.

:free-joint *joint-number* *Message*

This operation causes the current in the specified joint motor to be set to zero, thus causing the joint to go limp. The operation waits for the user to hit another character, then restores power to the motor. This operation can be used to loosen a joint, for instance, to move it back from being out of range. However, *use caution!!* The arm or something or someone may be damaged if joints 2 or 3 are freed allowing the heavy links 2 or 3 to fall.

6.4 Hand

:hand *amount* *Message*

Opens or closes the hand. *Amount* must be a number between 0 and 1, indicating a fraction of "wide open" to set the hand. [Note: Currently, only "closed", or 0, and "open", or non-zero, are recognized.]

This operation is *not*, repeat *not*, synchonized with arm motion. (As of this writing, that is.) To ensure that the hand is opened or closed before proceeding, the suggested procedure is to wait for motion to complete (`:wait-for-move`), open or close the hand, then wait for about a second before proceeding.

:close-hand *Message*

Causes the hand to close. Same as (`send` *puma* `:hand 0.0`).

:open-hand *Message*

Opens the hand. Same as (`send` *puma* `:hand 1.0`).

6.5 Compliance

:comply *force stiffness damping joint-damping frame* *Message*

This is the compliance debugging operation. The five parameter matrices are sent over to the wrist PDP–11, then the arm is held in place, compliantly. To stop the "move", send a `:comply-end` message (q.v.).

:comply-end *Message*

Ends a compliance debugging "move".

:comply-parameters *type matrix &key (wait-p t) (immediate nil)* *Message*

The `:comply-parameters` operation is used to set up new compliance parameter matrices in Cougar. *Type* is a keyword, designating which parameter matrix is being updated. The *immediate* flag, if non-nil, indicates that the matrix will be sent to Cougar right away; if nil, (not immediate), the matrix will be sent as soon as all currently queued motion segments have been completed (i.e. synchronized).

The defined *types* are:

`:force` The desired force—a matrix of six values, specifying forces in x, y, and z, and torques around the x, y, and z axes. The forces are specified in ounces; the torques are specifed in ounce-inches. Synonyms are `:commanded-force` and `:f`.

`:stiffness` The spring constant matrix. This is the diagonal of a six-by-six matrix. Synonyms are `:spring-constant`, `:spring`, and `:k`.

`:cartesian-damping` The damping matrix. This is the full six-by-six matrix. Synonym is `:b`.

`:joint-damping` The joint damping matrix. This is a six-element vector, one element for each joint. Synonyms are `:damping` and `:bj`.

`:frame-vector` The vector to the "center of compliance." This is a vector of three elements, the x, y, and z offsets from the center of the wrist. Synonyms are `:frame` and `:r`.

`:effective-weight` The effective weight of the load in the hand. This is a vector of four elements. Synonym is `:ewc`.

:initialize-compliance *type matrix ...* *Message*

This operation is equivalent to doing a `:comply-parameters` operation on all of the compliance matrices. Any matrices specified in the call are sent as specified. Those matrices not specified are set to zero, except for `:joint-damping`, which is set to the value of the symbol `puma:good-values-for-joint-damping`.

6.6 Guards

Guards and guarded motion is described in more detail in Section 8, page 45. This section presents the primitive operations in which guards are implemented. See the above section for a discussion of how these operations may be used.

:clear-guard *number* &key *(immediate nil) (wait-p t)* *Message*

Clears the guard numbered *number*. See `:set-guard`.

:set-guard *type number vector tolerance* &key *immediate (wait-p t)* *Message*

Sets the guard numbered *number*, which must be a number from 0 to 15 inclusive. *Type* is one of `:aborting` or `:continuing`, whether the arm will be stopped or not when the guard is tripped. The guard is set for *tolerance* ounces along *vector*, in the coordinate frame of the hand.

Immediate refers to whether the guard will be set right away, or after previously queued motion is completed. If *wait-p* is true, the operation will wait until the request is queued at the end of the motion requests before returning.

When the guard is tripped, a message is sent back to the Lisp Machine, indicating which guard tripped. The guard is automatically cleared.

6.7 Miscellaneous

This section lists several operations which are not generally useful when just operating the arm, but can be very useful for debugging programs and the Puma system itself.

:connection-open? *Message*

Returns T if there is a connection open to the Puma.

:deposit *value location* *Message*

Deposits *value* as a 16-bit number in the address *location*. Used only for debugging, and should be used with great care, if at all.

:examine *location* &optional *(count 1) 8bit-p* *Message*

Returns an array with a copy of the contents of the PDP-11's memory starting at *location* and continuing for *count* words. A second value is returned, which is the number of words actually examined. (This value may differ from the requested number because of NXM's.) If *8bit-p* is set, then the result is returned as an `ART-STRING` array, otherwise as an `ART-16B` array. Note that in the former case, *count* still refers to words, even though the result is returns as an array of bytes. As a convenience, if *count* is not supplied, the single word examined is returned as a fixnum, rather than a one-element array. For example:

```
(send *puma* :examine #o62100 #o40)
#<ART-16B-32. 74132631>
32.
(listarray *)
(61527. 0. 2615. 16848. 65047. 1000. 63799. 3392.
   64311. 53034. 5568. 34612. 5569. 28844. 5570.
   28868. 5571. 6. 62736. 63505. 63799. 3364. 63506.
   32454. 135. 7616. 16798. 2615. 2804. 2551.
   32. 2551.)
(send *puma* :examine #o62100)
61527.
```

Many interesting addresses can be retrieved from the PDP–11 by using the `:get-parameter` operation (q.v.).

:flush-queues *&key (rip-queue t) (signal-waiters t)* *Message*

This operation will cause any pending requests to be removed from the queues. This might be useful as part of recovery from errors. The argument *rip-queue* says to remove entries from the requests-in-process-queue, as well as the user-request-queue. If *signal-waiters* is true, then any processes waiting on the requests will get an error signal.

:get-parameter *name &optional (return-type :decode)* *Message*

A general purpose operation to retrieve various bits of interesting information about the state of the arm. The *name* is a keyword name of the data desired. *Return-type* is a keyword indicating how you want the data. It may be one of `:list`, `:array`, or `:decode`. The `:decode` return type attempts to be intelligent about what to return: a keyword, if appropriate, a number, a list or an array. See the operation `:set-parameter` for a list of the recognized parameter names.

:notify-trivial-errors? *Message*

Returns T iff so-called "trivial" errors are presented to the user as notifications, rather than the background process entering the debugger.

:send-self-and-wait-until-done *operation &rest args* *Message*

Same as
`(progn (lexpr-send puma operation args)`
`(send puma :wait-until-no-requests-pending)).`
(See page 39.)

:set-notify-trivial-errors? *new-value* *Message*

This operation allows the user to specify whether "trivial errors" are presented to the user as notifications, or by the background process entering the debugger. The errors which are regarded as "trivial" may be specified by the `:set-trivial-error-codes` operation.

:set-parameter *name value &key wait-p* *Message*

A general purpose operation to alter various parameters concerning the operation of the Puma 11. *Name* is a keyword specifying which parameter to alter (see below), *value* is the new value, and *wait-p* specifies whether to wait until the Puma 11 responds before returning.

The parameters which can be examined or altered are as follows:

`:maxacc` The maximum acceleration of the six joints. This is a six-vector of floating point numbers.

`:wsttim` The number of servo-cycles (of 16ms) between times the wrist is queried for New Theta Dot values during compliance. Normally 1, this parameter is a single 16-bit word.

`:complying-joints` This parameter is a vector of six elements, corresponding one to each joint. If the element is zero that joint will not comply.

:sync-errors This is a count of the number of communication synchronization errors reported by the wrist PDP-11. It is a single 16-bit word, and should normally be zero.

:reset-meters :meter-pointers
When read, as :meter-pointers, this returns a vector of all the meter pointer structures in the PDP-11. Each structure contains 4 16-bit words: The beginning address of the buffer, the ending address, the current pointer address, and a code number identifying the meter. When written, as :reset-meters with no data, this parameter sets all the meter pointers to the beginnings of their respective buffers.

:buffer-pointers This read-only parameter returns the address of the beginning of the block of pointers to the wrist communication ring buffers. The block contains four words: the input top pointer, input bottom pointer, output top pointer, and output bottom pointer.

:buffer-length This read-only parameter returns the length of the wrist communication ring buffers. Also known as :buffer-size.

:interrupts-missed Returns a count of the number of times the real time clock overflowed while we were still processing the previous interrupt. Under normal operation, this should be zero. Also known as :intmis.

:metered-joint A number from 1 to 6 inclusive. The joint specified by the number will be the one whose angles and velocities are stored in the various meter buffers.

:cycle-time This parameter controls the servo cycle time of the Puma 11. Normally 16 milliseconds. Should be change only with great care and very good reason.

:miflgs The interrupt process flag word. Marginal usefulness.

:version Read-only. Returns a 9-vector of version numbers of the software modules in the PDP-11. This is the same information as the :software-version data from the :describe-manipulator operation.

:move-type This keyword parameter specifies the interpolation scheme to be used for moving between setpoints. The recognized keywords are :5D for the fifth degree polynomial, and :2D for linear with quadratic corners.

:transition-time The number of servo-cycles in the transition between segments for the quadratic corners. More precisely, this is the number of cycles on either side of the setpoint to the beginning or end of the transition. Also known as :d2trnt.

:speed-factor This is a floating point number, and is the speed at which the arm should move. (send *puma* :set-parameter :speed-factor *value*) has the same effect as (send *puma* :speed *value*).

:vffon A boolean value, 1 or 0. Specifies if the velocity feed-forward calculations should be performed. Also known as :velocity-feed-forward-on-p

:boverk The velocity feed-forward constant. Also known as :feed-forward-constant.

:dc-correction This is a vector of six elements, and represents the experimentally discovered DC correction to the joint motor currents required for accurate positioning. Also known as :pmdcof.

:ppon A boolean value, 1 or 0, specifying whether the predict-preview compensation calculations are enabled. This feature does not seem to provide significantly better tracking, and has a tendency to make the arm control unstable. It is very sensitive to the values of K^*, the constant for the calculations. Its use is not recommended. Also known as :predict-preview-compensation-on-p.

:kstar The K^* constant for the predict-preview compensation calculation. Also known as :predict-preview-constant.

:set-trivial-error-codes *codes* *Message*

Specifies which of the three-letter error codes the Puma will consider as trivial for the trivial error notification feature. *Codes* should be a list.

:solve-backwards *joint-angles* *Message*

Returns the transformation representing the position in space at which the hand would be with the joints at the specified angles.

:solve-forwards *transform joint-angles* *Message*

Get the LSI-11 to call the solution code on *transform* and *joint-angles*—the joint angles pretend to be where we are now so we can force it to choose configurations. Useful mostly for debugging the solution code. Returns a set of joint angles.

:trivial-error-codes *Message*

Returns the list of three-letter error codes the Lisp Puma object considers trivial enough to not require user intervention in recovery.

:wait-until-no-requests-pending *Message*

Waits on the Lisp end until there are no outstanding requests to be processed. Unlike `:wait-for-move`, requests may still be processed by the PDP-11 while this operation is waiting.

:wrist-deposit *value location* *Message*

Like `:deposit`, but refers to Cougar instead of Puma.

:wrist-examine *location* &optional *(count 1.) 8bit-p* *Message*

Like `:examine`, but refers to Cougar instead of Puma.

:write-gauge-offset-vector *vals* &key *immediate wait-p* *Message*

Stores *vals* in Cougar as the calibrated wrist strain gauge offsets. *Vals* should be a list or array of eight 16-bit integer values.

6.8 Requests (low level)

This section presents the most primitive operations available to the user to manipulator the Puma arm. These are not generally necessary, as the above described operations are sufficient for most uses. However, any Puma operation may be performed by using just these operations to create and send a request message the the Puma PDP-11.

Section 5, page 26, gives a brief description of the protocol used to communicate between the Lisp Machine and the PDP-11, and how the Lisp Machine manipulates "requests."

:request *request-code* &optional *data* &key *(response-type :single) wait*
data-extractor return-request request-interrupt *Message*

This is the primitive which is used to issue a request over the network to the Puma 11. The arguments *request-code*, *data*, *response-type*, *data-extractor*, and *request-interrupt* are as in `:make-request` (see page 40). If *return-request* is specified, then the actual request is returned from this message. Otherwise, if results are expected from the request, those results are returned; if not, then `T` is returned.

This operation will wait until the request has been responded to if either of the following conditions is met:

1. The *wait* argument is supplied, in which case it should be a string to display in the who-line, or `T`, which means to use the name of the request in the who-line.
2. The *wait* argument is not supplied and the request is expecting data in the response. This is indicated by a non-nil `data-extractor` field in the request, either supplied in the call to the `:request` operation or defaulted by the request code.

By supplying `:return-request t :wait nil`, the program may issue a request expecting data, but go on to other computations and come back later to retrieve the data. The program may wait for the data by calling (`manip:wait-for-response` *request &optional who-state*), or may check for the response having been received by calling (`manip:request-data-ready?` *request*). In the latter case, the program should call `manip:wait-for-response` before getting the data out with (`manip:request-data-drop-off` *request*) because the former function performs some needed error checking.

The `:request` operation is implemented in terms of `:make-request` and `:send-request`.

:make-request *request-code* &optional *data*
&key *(response-type :single) data-extractor request-interrupt* *Message*

Creates a manipulator request, suitable for passing to the `:send-request` operation. The *request-code* is a symbolic name for a manipulator protocol operation, such as `DESCRIBE` or `MOVE`. *Data* is whatever arguments the operation requires, such as a transform, a speed factor, or a joint angle vector. *Data* may be one of the following:

A fixnum The number is interpreted as a single 16-bit integer.

A flonum The number is interpreted as a single 32-bit floating point number.

An array The array should contain numbers, which are interpreted depending on the type of array, floating or fixed point numbers. `ART-Q` type arrays are interpreted as floating point numbers (3600 only).

A list Each element of the list is recursively interpreted, and appended to the packet going to the PDP-11.

The *response-type* and *data-extractor* values almost always have reasonable default values. See the "An Implementor's Guide to the Lisp Manipulator System" for more details.

Request-interrupt allows the programmer to specify an action to be taken as soon as the response for the request comes back from the PDP-11. *Request-interrupt* may be any of the following:

A process A `request-response` condition is signalled in the process. Instance variables in the condition store the request, the response packet, and a flag indicating whether an error occurred in the processing of the request.

A form The form is evaluated. Any return values are discarded. The special variables `manip:manipulator`, `manip:request`, `manip:response`, and `manip:error-p` are bound for the evaluation.

A function The function is applied to four arguments, the manipulator, request, response packet, and the error flag.

A symbol If the symbol has a definition, it is applied to the same for arguments as above. Otherwise, it is ignored.

With the exception of the condition being signalled in the specified process, all these actions are interpreted in the background process.

:send-request *request* &key *wait* *Message*

Queues a request, as is created by `:make-request`, to be sent to the Puma 11. If *wait* is non-nil, this operation will await the response from the manipulator, and return whatever data is in the response; otherwise, this operation will return immediately. When waiting, if *wait* is a string or symbol other than T, it will be displayed in the who-line.

The `:send-request` operation is implemented in terms of `:queue-request`.

:queue-request *request* *Message*

Finally, the `:queue-request` operation is the most primitive operation available to the user to

send a request to the Puma PDP-11. The *request* must be created with either the :make-request operation or the make-manipulator-request macro; it is placed on a queue for the background process to pick up and send off to the Puma PDP-11. This operation returns immediately upon queueing the request; any waiting or retrieving of data must be done by the caller.

7 Error System

Error handling for the Puma is a bit of a problem, due to the asynchronous nature of the system. An error may be detected in response to a user request, but the user process will have continued its computation, and may not be prepared to handle the error condition. In some cases, it may be easy to determine if the error should be signalled for the user process, in others it may not be so easy. This chapter describes the error system designed in an attempt to handle this situation.

7.1 Arm Errors

There are four types of errors which can occur while using the Puma. The first type of error is signalled by the PDP-11 in response to a user request. The PDP-11 can detect a condition with the arm which prevents it from satisfying the user request, such as the requested joint angles being out of range, or the specified point being outside the workspace. In this case, the PDP-11 would reply to the Lisp Machine with an error response in lieu of the normal response to the request. If possible, this error would be signalled to the process which originated the request.

The second type of error is signalled by the PDP-11 independent of any user requests. These errors would occur if there were a condition which prevents further operation, but is not necessarily associated with a specific user request. For example, suppose motion were requested. The motion command would cause the trajectory segment to be queued, then would be responded to immediatedly. The motion would occur asynchronously with the command response. In fact, several requests may be processed before the motion actually may begin. Suppose, now, the arm power were off, preventing the actual motion. An error cannot be sent in response to the original request. The PDP-11 responds with an "asynchronous" error, which is then brought to the attention of the human user of the Lisp Machine.

These two types of errors are considered "arm" errors, since they represent conditions in the arm control processing which prevent further operation until the conditions are remedied. Both of these types of errors may be divided into non-fatal errors and fatal errors. Non-fatal errors are those which (presumably) may easily be corrected, and allow the user to proceed. These would include such things as joint angles out of range, arm power is not turned on, communication with the wrist PDP-11 has ceased, etc. In these cases, operation of the arm may continue after clearing up the problem, or possibly ignoring and avoiding the problem. The Puma system goes to some effort to signal these errors in such a way that the program may correct the problem and proceed without human intervention, if possible.

Conditions may arise which would make it not possible to continue operation of the arm under any circumstances. For instance, if communication with the arm controller hardware were to cease, it would become impossible to continue the program. In these cases, the PDP-11 will signal a fatal error, and close the network connection to the Lisp Machine. Fatal errors are not handed to the user program by the Puma system. The human user is alterted and processing of further requests ceases.

All errors sent from the PDP-11 to the Lisp Machine contain a three letter code abbreviation for the error, along with a severity indication and a message describing the error. The three letter error code may be used by the user program to specify what errors it is prepared to handle. See Section 7.4, page 43.

7.2 System Errors

The remaining two types of errors are considered "system" errors; they represent conditions in the system software which would prevent further computation, as opposed to problems with the arm. The first of these types of errors occurs when the Puma system detects something wrong with the user's request, with an operation the user is attempting, or with its own internal operation. In such a case, the program which caused the condition will receive the error. If the error occurs in the background I/O process, this is a bug, and should be reported. If the error occurs in the user program, then chances are that the user program has attempted something illegal or malformatted.

The final type of error is simply a Lisp error. These errors are those that the underlying Lisp Machine software detects and signals. Again, if they occur in the background process, it is a bug and should be reported.

"System" errors always occur because of a condition on the Lisp Machine itself. "Arm" errors always occur in response to a condition on the other side of the connection, in the PDP-11.

7.3 Error Signalling

The error system is built on the Lisp Machine error facility (see the Lisp Machine Manual). Thus, error conditions are signalled as flavor instances, the particular flavor indicating the type of error. This and the following sections assume the reader has at least some familiarity with flavors in general and Lisp Machine condition handling in particular.

As hinted at in the introduction to this chapter there are some issues involved with the asynchronous nature of Puma non-fatal errors. The user program may issue a request of the Puma PDP-11, and not wait for the response. It may issue several such requests before looking for a response to any of them. An error occuring in the processing of one of the requests may make it undesirable, dangerous, or impossible to satisfy later requests. However, if the user process cannot be depended upon to check any request for a possible error, or do so in a timely manner, then some means must be used to bring the error condition to the attention of that process or to the user.

Thus, the following goals were set forth for the design of the Puma error system:

- The process making the request should be given an opportunity to handle an error occuring in the handling of the request. This requires that process informing the background process, who fields the errors, that it is prepared to do so.
- A process should be able to identify itself as being able to handle errors in the processing of a request, whether or not it is the process who made the request. This would allow a group of cooperating processes to delegate one process to be the error handler.
- Processes waiting for the response to a request must not wait forever if the request resulted in an error. They must be notified.
- If no process can be found to handle the error condition, it must be brought to the attention of the user, so that any appropriate action may be taken before proceeding.

The following mechanism was developed to meet these goals. When the background process receives an error packet from the PDP-11, it matches the error with the request to which it corresponds. If the `manip:request-error-signal` slot of the request is non-NIL, and is a process, then that process will get the error. Otherwise, if the process who made the request (the `manip:request-requesting-process` slot) has specified that it is willing to handle the particular error code which has been received (by a mechanism to be described shortly), then that process will get the error. If neither of the above are the case, then the

error is given to the process which is waiting for the response, if any. Finally, if no process can be found to which to give the error, then the background process itself will get the error.

The phrase "get the error" used in the preceding paragraph means that an error condition will be signalled in that process. The particular error flavor to be signalled depends on the error code found in the error message (as described in the section Error Flavors, below), but will always be built on `manip:arm-non-fatal-error`. For processes other than the background process, the error is signalled by sending the process an `:interrupt` message; in the background process, the error is signalled directly. The error is signalled with the `signal` function rather than the `error` function so that a handler can do what it can to correct the error and proceed without disturbing the computation in progress.

When the background process itself signals the error, then, of course, it is stopped, an cannot process further requests, responses, or errors until the user takes some action to proceed or restart the process. Certain errors are always signalled in the background process; these include asynchronous errors—errors not belonging to any request, fatal errors, and all Lisp errors detected in the background process.

Special care is taken for processes waiting for the response to a request, so that an error will be signalled if they happen to try to wait for the request when an error reply already arrived before or while they wait. Also, if the process which is waiting also happens to be the one that "gets the error," then it does not get another error by virtue of waiting.

7.4 Error Flavors

The Puma system has defined a family of error flavors for the various types of error conditions that can develop. These are listed in detail in a later section. For the moment, it is only necessary to describe the general hierarchy of the family. All error conditions relating to either the arm or the Puma system software are built on `manip:manipulator-error`. Arm errors in particular are built on `manip:arm-11-error`; more specifically, non-fatal error responses are of type `manip:arm-non-fatal-error`. "System" errors are of the flavor `manip:system-error`.

All non-fatal arm errors include a three letter code, which can be used to identify the error. These codes may also be used to select which of the non-fatal errors the user wishes to handle (see below). A list of the error codes currently used in the Puma is provided in a separate document. The error code can be examined from any `manip:arm-11-error` object with the `:error-code` operation.

Naturally, all the normal condition handling techniques may be used for the Puma errors. However, there is a problem when the process desiring to handle the errors needs to inform the background process that it is indeed interested in handling the errors, and which errors it wishes to handle. For this purpose, the following special forms have been defined.

manip:def-arm-error-flavor *flavor-name error-code...* *Special form*

The *flavor-name* is defined as a mixin flavor. The specified *error-codes* are associated with this *flavor-name* to indicate which PDP–11 errors this flavor is to be used for. Note that it works to associate the same error code with more than one error flavor. When any of the error handling forms described below are used to set up the error handling, the flavors specified imply which error codes are to be handled by the process.

manip:error-bind *bindings body...* *Special form*

See below.

manip:error-bind-if *condition bindings body...* *Special form*

See below.

manip:error-case *(var...) form clause...* *Special form*

These special forms are analagous to the forms after which they were named, `condition-bind`, `condition-bind-if`, and `condition-case`. In addition to the functionality of the condition

handling, they set up an environment such that the background I/O process recognizes that any errors with the error codes associated with the flavors specified are to be handed to this process. If more than one flavor is used which has been associated with the error code, then all the flavors are mixed together to form the condition object. Thus each handler will have a chance at the error.

manip:into-debugger-errors *(error-code...) body...* *Special form*

This form allows a program to specify error codes for which it has no handler, but for which it wishes the background process to hand it the errors. This allows the program to go into the debugger and let the user take care of the error without having the error signalled in the background process. Note that this form requires the three-letter codes themselves—not flavor names.

Perhaps an example would help clarify the use of these forms:

```
(manip:def-arm-error-flavor joint-out-of-range JOR TOR)

(manip:def-arm-error-flavor arm-power NAP)

(defun test-function (arm start finish other)
  (manip:error-case (e)
        (progn (send arm :move start)
               (send arm :set-joint-angles other)
               (send arm :move finish)
               (send arm :wait-for-move))
    (joint-out-of-range (format t "We got an error,  A. What now?" e))
    (arm-power (format t "Turn on the power, then try it again."))
    (:no-error (format t "Test completed."))))
```

Note that the protected form needed to wait for the motion to complete. This is a problem with the error handling, because the process cannot leave the protection of the error handlers until the operations being protected have all completed.

7.5 List of Error Flavors

manip:manipulator-error *Error Flavor*

This is the basic manipulator error flavor. All manipulator errors are built on this one. It accepts the message `:manipulator` which returns the manipulator object which got the error.

manip:pdp-11-error *Error Flavor*

This flavor is mixed in to any error which originated on the PDP–11. It will accept the messages `:error-code`, which returns the three-letter code as a symbol in the `MANIPULATOR-ERROR-CODE`, or `MEC` package, `:error-severity`, a single letter code indicating the severity of the error, `:error-message`, the string description of the error, and `:request` the request to which the error corresponds, if any.

manip:manipulator-arm-error *Error Flavor*

A mixture of `manip:manipulator-error` and `manip:pdp-11-error`.

manip:manipulator-lispm-error *Error Flavor*

This flavor is mixed in to any error occurring on the Lispm side of the connection.

The following flavors are all built on `manip:manipulator-arm-error`, and thus are PDP–11 errors.

manip:arm-fatal-error *Error Flavor*

This flavor is signalled when the Lisp Machine receives a fatal error from the Puma. It is signalled in the background process only.

manip:arm-non-fatal-error *Error Flavor*

All non-fatal errors reported by the PDP-11 include this flavor.

manip:error-object-in-data-ready *Error Flavor*

This error is signalled whenever the user attempts to wait on a request which has already received an error response, and thus `manip:wait-for-response` finds the error in `manip:request-data-ready?`.

These flavors are built on a mixture of `manip:manipulator-error` and `manip:manipulator-lispm-error`.

manip:system-error *Error Flavor*

This represents a error in the Puma system, and always represents a bug of some sort. These errors should be reported to the maintainers.

manip:command-error *Error Flavor*

This flavor indicates that an error was detected in the formation or use of one of the Puma system requests, operations, or functions. The most probable cause of an error of this type is malformatted data.

manip:unhandled-operation *Error Flavor*

This is built on the `manip:system-error` flavor, and indicates that an attempt was made to send a request to the Puma with a request code which was not in the list of handled operations. The following proceed options are defined: `:go-ahead`: send the request anyway, `:do-describe`: ask the Puma again which operations it can perform, to verify that there is a discrepancy, `:go-ahead-remember`: send the request, and add the code to the list of handled codes, and `:punt-this-one`: don't send the request, but proceed without any other special action.

manip:unknown-manipulator-code *Error Flavor*

Built on `manip:system-error`, this condition is signalled when an undefined symbolic request code was used, as in, for example, `(send *puma* :request 'manip:raedy)`, where "ready" is misspelled. The message `:bad-one` will return the incorrect symbol. This condition allows two proceed types: `:new-code`: takes a new symbol as an argument, and tries to use that symbol for the request code, and `:new-code-save`: takes a number as an argument, and defines the bad code to have that number as the actual request code value.

manip:request-flushed *Error Flavor*

This condition flavor is actually built on the flavors `manip:manipulator-lispm-condition` and `manip:manipulator-condition`, which, in turn, are built on `condition` instead of `error`. Thus, if no handlers are specified for this condition, no action is taken.

This condition is signalled for any process which is waiting on a request which is forcibly removed from its queue. It is also placed in the `manip:request-data-ready?` slot of the request.

manip:request-response *Error Flavor*

This is similar to `manip:request-flushed`, being built on `manip:manipulator-condition`. It is used for the `manip:request-interrupt` feature of requests. See the `:make-request` operation, Section 6.8, page 40.

8 Guards

manip:with-guard-set *(puma options...) forms...* *Special form*

This executes *forms* with a guard set on the *puma* manipulator. The guard parameters are specified by the *options*. Each option is a keyword, possibly followed by an argument. The defined options are:

:name *name* Here, *name* is a name for the guard. This may be useful when more than one guard is being set, in conjunction with `guard-wait`.

:vector *vector* Defines the direction in which the force is to be monitored. *Vector* should be a list or array with six elements.

:tolerance *force* Specifies the force at which the guard will trip. Let $\mathbf{n}$ be the force sensed on the hand, $\mathbf{v}$ be the vector specified by the `:vector` option, and t be the tolerance; then the guard will trip when $\mathbf{n} \cdot \mathbf{v} > t$.

:x *force*, **:y** *force*, **:z** *force*, **:xt** *torque*, **:yt** *torque*, **:zt** *torque* These are a shorthand for specifying a `:vector` and `:tolerance`, when the desired vector is along one of the coordinate axes, or a torque about one of the axes. Only one of these may be specified, and `:vector` and `:tolerance` may not be specified. (This restriction may be lifted in the future.) For example, `:x 55` is equivalent to `:vector '(1 0 0 0 0 0) :tolerance 55`.

:trip-form *form* Specifies a *form* to execute if the guard should be tripped. When the guard actually gets tripped, this form is executed and the `manip:with-guard-set` form is exited. Whatever values *form* returns are returned by `manip:with-guard-set`. If no `:trip-form` is specified, `manip:with-guard-set` returns `nil`. If the guard is not tripped, the form passes on the values returned by the last form in the body.

:continuable Specifies that the guard does not stop arm motion, but merely alerts the Lisp Machine that a guard was tripped.

:no-continuable Explicitly specifies the default, that is, to stop the arm and abort the trajectory when a guard is tripped.

manip:guard-wait &optional *guard-name* *Special form*

Waits until the guard named *guard-name* is tripped. Waits forever if the guard is never tripped. If *guard-name* is not supplied, then the innermost guard of a `manip:with-guard-set` is the guard waited for.