Trident Robotics and Research, Inc.

PUMA100

User's Manual

2516 Matterhorn Drive, Wexford, Pennsylvania 15090-7962
(412) 934-8348

This device is not intended for use in life support equipment and should not be used in any medical or other application where intermittent malfunction or failure may directly jeopardize the health or well-being of an individual or individuals without adequate safeguards.

Address comments concerning this document to:

Trident Robotics and Research, Inc.
User Documentation Dept.
2516 Matterhorn Drive
Wexford, PA 15090-7962
(412) 934-8348

Revised: September, 1997

PUMA100 VAL Emulation Library

Version 1.22

1. Installation

1.1. Hardware

Install the TRC007 in slot IP1A of the TRC100. This corresponds to slot 3 in the "TRR_PUMA.CFG" configuration file. Connect the 50-pin cable to the TRC004 to P13 on the TRC100. Configure the TRC100 for 64-byte address space (see TRC100 User's Manual) and, with the personal computer power off, install the TRC100 in an empty slot. The 50-pin ribbon cable should be routed through the hole of an adjacent empty slot.

1.2. Software

To initiate file and console support for the TRC100, run the TSR "srvr100.exe." This is invoked with "srvr100 <base address>."

After power up or warm or cold reset of the TRC100, there is a 10-20-second delay while the system performs self-test and checks serial port A for activity. If nothing is found on the serial port, the TRC100 waits to download an executable over the PC bus. The program "trident.exe" downloads S-record files to the TRC100. The file "TRIDENT.CFG" establishes the base address and other parameters of the TRC100, as well as the default executable file. Invoking "trident" or "trident <filename>" begins the download sequence.

With these programs running on both the host PC and the TRC100, the VAL emulation software is ready to receive commands.

2. DOS Commands

potcal
calib
where
teach
speed
do_move
do_moves
do_dmove
do_ready
do_open
do_close
do_relax

3. Library Functions

The header file "val.h" contains the function prototypes for the simulated VAL C library, some macros, and two sets of #define'd function return values. The functions and macros are described in subsequent sections. The function return values consist of PC return values:

I_OK
I_ERROR
I_RANGE
I_ABORT
I_TMOUT

and TRC100 return values:

CMD_OK
BAD_CMD
BAD_SEQ
BAD_DATA
BAD_FILE
BAD_POWER
BAD_EXEC

The return values I_OK and CMD_OK are equivalent. All other values are unique.

3.1. Program Initialization

Every simulated VAL program must call a software initialization function before issuing any simulated VAL commands. The prototype for this function is:

```
int initialize()
```

This function expects to find the configuration files "trident.cfg" and "trr_puma.cfg" in the local directory. It returns either I_OK or I_ERROR where I_ERROR indicates the inability to open one of the above files. In this event, a diagnostic message is printed to the console if in text mode.

3.2. Variable Initialization

VAL defines two types of location variables: precision points and transformations. Precision points are highly repeatable but are specific to a particular class of manipulator (e.g. Puma 560). Transformations, on the other hand, are not specific to a particular class of manipulator and are portable across robots and applications. Both types of location variables are stored in the location structure defined in val.h.

To create a precision point, allocate space for a location variable and call:

```
location *ppoint(location *loc_p, float j1, float j2, float j3, float j4,
            float j5, float j6)
```

where the jX values are the six joint angles in degrees. An example of proper use of this function is contained in the following code fragment:

```
#include "val.h"

location point1;        /* allocate space for the location */
```

```
ppoint(&point1, 0.0, -100.0, 115.0, 0.0, 87.3, 46.8);
```

This function performs no range checking and returns **loc_p**.

To create a transformation, allocate space for a location variable and call:

```
location *trans(location *loc_p, float x, float y, float z, float o, float a,
            float t)
```

where x,y,z describe the position of the wrist in millimeters and o,a,t describe the orientation of the wrist in degrees. An example of proper use of this function is contained in the following code fragment:

```
#include "val.h"

location point1;

trans(&point1, 100.0, -100.0, 300.0, 0.0, 87.3, 46.8);
```

This function performs no range checking and returns **loc_p**.

3.3. Location Functions

The function set() copies one location point into another location point.

```
int set(location *dest_p, location *src_p)
```

Storel() and loadl() store and load location variables to/from file, respectively.

```
int storel(char *fname, location *loc_p)


int loadl(char *fname, location *loc_p)
```

The function here() gets the current location of the robot and places it in the location variable provided.

```
int here(location *loc_p)
```

The function where() prints the current location of the robot to the console. It is incompatible with Windows.

```
int where()
```

3.4. Move Functions

All move commands are affected by a small number of operating parameters. These parameters include the speed of motion, the configuration of the robot, and the blending of adjacent motion segments. These parameters are set independently of the individual move commands and remain in effect until explicitly changed.

The function speed() sets the speed of all subsequent move commands *including subsequent programs*. Acceptable values for spd range from 0.01 to 327, but all values less than or equal to one are equivalent.

```
int speed(float spd)
```

The functions valEnable() and valDisable() set VAL switches. The only VAL switch currently supported is the “CP” switch for continuous path motion. If this switch is disabled, the robot completely finishes one move before beginning the next. If this switch is enabled (the default), the robot blends the end of one move with the beginning of the next, producing a smoother transition, but an inexact trajectory.

```
int valEnable(int switch)
```

```
int valDisable(int switch)
```

where “switch” can only be the #define’d value “CP”.

Another “switch” that is not VAL compatible is the “TRC_ALTER” switch. By default, this is disabled. If enabled, a “move” command will interrupt any pending “move” commands, causing an immediate replanning of the trajectory to the new goal point. This is similar to the VAL “alter” mode except that strict timing between goal updates is not required. Instead, intermediate goal points are inserted automatically, as required by the current speed setting.

Move() moves the robot to the location specified by the pointer loc_p in joint-interpolated mode. This produces unpredictable but controlled motion of the end effector through space as each joint moves the shortest distance to its destination. Currently, loc_p can be either a precision point or a transformation.

```
int move(location *loc_p)
```

Return values include both PC and TRC100 values with the following meanings:

I_RANGE	location outside joint limits
I_ERROR	location not defined properly
BAD_SEQ	arm power not on

The move() command, like all motion commands, returns immediately after initiating the motion. It does not block until the motion is complete. However, the move() command will block if another motion command is in process. Upon completion of the pending motion command, the new motion command will automatically unblock.

To force blocking until the completion of a pending motion command, use the valBreak() instruction.

```
int valBreak()
```

The precise time of unblocking is determined by the state of the CP switch. If continuous path motion is enabled (default), unblocking occurs slightly before the robot has settled at the goal location, when approximately ninety percent of the trajectory is complete. If continuous path motion is disabled, unblocking occurs when the goal has been reached.

Moves() moves the robot to the location specified by the pointer loc_p in straight-line mode. The end effector follows a straight line through space while the orientation rotates to the desired goal pose. Currently, loc_p can be either a precision point or a transformation.

Care must be taken to avoid singularities in the robot's workspace. For instance, moves() can not be executed through a complete extension of joints 2 and 3 or the alignment of joints 4 and 6.

```
int moves(location *loc_p)
```

Return values include both PC and TRC100 values with the following meanings:

I_RANGE	location outside joint limits
I_ERROR	location not defined properly
BAD_SEQ	arm power not on

Occasionally, it is necessary to move the robot with respect to its workspace after locations have been stored and programs developed. If the locations are stored as transformations, it is easy to compensate for this with the base() command. Upon initialization, the origin of the robot's coordinate frame is reset to the VAL origin at the intersection of the shoulder and waist axes. A call to base():

```
int base(float dx, float dy, float dz, float dr)
```

with **`dx`**, **`dy`**, and **`dz`** representing the motion of the robot's frame with respect to the original VAL frame, in millimeters, relocates the position of the robot with respect to previously-defined transformations. **`dr`** represents the incremental rotation of the base around the z-axis, in degrees.

All four of the arguments to the base() command are cumulative from the most recent initialize() command and are defined relative to the VAL world frame. Precision points are not affected by the base() command.

To move the end effector simultaneously with the arm, use movet() and movest() for joint-interpolated and straight-line motions, respectively.

```
int movet(location *loc_p, float hand)
```

```
int movest(location *loc_p, float hand)
```

The argument "hand" specifies the opening of the end effector. Currently, only pneumatic end effectors are supported so if hand is positive, the end effector opens. If hand is zero or negative, the end effector closes. Return values are identical to move() and moves().

Ready() executes a move to the ready position (straight up). The ready position is defined as the joint angles: 0, -90, 90, 0, 0, 0 in degrees. Return values are identical to move().

```
int ready()
```

Dmove() executes a "delta move." The jX values are increments from the current joint positions in degrees. Return values are identical to move(). This is not a feature of VAL-II and is provided only for convenience.

```
int dmove(float j1, float j2, float j3, float j4, float j5, float j6)
```

Draw() executes a "delta cartesian move" in translation only. The dX values are increments from the current cartesian positions in millimeters. Return values are identical to moves(). This is a feature of VAL and was removed from VAL-II. It is provided only for convenience.

```
int draw(float dx, float dy, float dz)
```

Teach() allows teaching of points via the keyboard. The numbers "1" through "6" select a joint and the keys "+" and "-" increment and decrement the joint position. The keys "d" and "h" double and halve the step size, respectively, increasing or decreasing the effective speed of motion. Pressing <enter> returns I_OK while pressing "a" returns I_ABORT.

Teach() is incompatible with Windows.

```
int teach()
```

3.5. Gripper Functions

The following functions immediately open, close, and relax the gripper pneumatic lines:

```
int openi()
```

```
int closei()
```

```
int relax()
```

3.6. Calibration

There are two types of calibration that must be performed: power-up calibration and pot calibration. In general, these functions are executed from the DOS command line rather than a user executable. However, subroutine calls have been provided. These calls are not compatible with the Windows OS.

Power-up calibration is the day-to-day process of initializing the high-precision incremental joint position encoders with the help of low-precision potentiometers at each joint. The potentiometers maintain absolute position information through power cycles while the encoders "forget" where they are every time the robot is powered down.

To perform power-up calibration, first a lookup table is loaded from the file "puma.pot". This lookup table contains an accurate encoder value for every corresponding value of the potentiometer. Then the joints are moved to find the "index pulse" of each encoder. The lookup table provides the exact encoder value which is loaded into the encoder counter.

The prototype for this function is:

```
int calibrate()
```

The function first asks for confirmation. If the arm is already calibrated and the command is aborted, it returns I_ABORT. If the command is aborted and the arm is not calibrated, it returns I_ERROR.

If calibration is not aborted and the lookup table is not loaded, arm power is turned off while file access is underway. When the "Ready to Calibrate" message appears, arm power is enabled. Once the "Arm Power On" button is pressed, motion begins immediately.

Some runtime errors can cause the arm to shutdown without stopping the calibration routine. If this occurs, simply turn arm power back on.

Pot calibration is the process of generating the lookup table and storing it in the file "puma.pot". In order to do this, potcal relies on a set of six "magic numbers." These magic numbers must be identified the first time potcal is executed and any time major maintenance is performed involving the removal or adjustment of a motor, encoder, or transmission element. Once these numbers have been identified, they should not be changed unless a major maintenance procedure is performed.

The function prototype is:

```
int potcal()
```

Potcal() expects to find the file "potcal.cfg" in the local directory. The format of this file must not change, including the number and location of blank lines and comments. A flag in this file is set to "1" to find the magic numbers automatically, or "0" for normal operation.

To perform potcal, it is necessary to position the arm in the ready position (straight up). See the Unimate manual for the exact definition of the ready position. If magic numbers are to be found, it is of the utmost importance that the arm be positioned **very accurately** in the ready position. The magic numbers determine the kinematic accuracy of joint positions and have a profound impact on straight-line motions and transformations. Subsequent executions of potcal that rely on existing magic numbers must begin in the ready position, but initial accuracy need only be in the realm of +/- 1 degree. Higher accuracy is demanded to extract magic numbers!

Upon starting potcal, turn on arm power and each joint will be moved through its full range of motion in sequence. The actual range of motion of each joint is specified in the file "potcal.cfg". The workspace must be clear of obstructions as the range of motion of each joint determines the range over which "calibrate()" can be executed successfully.

4. <u>Administrative Functions</u>

A small number of commands have been provided for system administration. To use these, include the header file “valadmin.h”.

The function logVars() logs certain PUMA variables to the text file “test.dat”. Logging is turned on if “flag” evaluates to true. Logging is turned off and the data is saved if “flag” evaluates to false. PUMA arm power will also turn off while the data is being saved to disk.

```
int logVars(int flag)
```

LogVars() returns a TRC100 return value.

The function TRCversion() returns the current version number of the VAL emulation library. If “flag” is zero, it also prints the version number to the console. The flag must be non-zero for Windows programs.

```
float TRCversion(int flag)
```