TROL
TU Vienna Robot Control Library
V 2.2
User Manual

Thomas Melzer
email: melzer@prip.tuwien.ac.at

April 27, 1998

Chapter 1

Introduction

Programming of the CRS-A465 industrial robot has normally to be done in the RAPL robot control language. While being well suited for typical industral applications, RAPL lacks the richness of data types and control structures of modern high level languages, features which will be needed when implementing more complex (e.g. 3D vision or real time control) tasks. For this reason, we have designed and implemented an A465 control library, which enables the programmer to control the robot from within his application written in 'C(++)'.

This document describes how to install and use TROL and is mainly intended for end users. Please refer to the "TROL Maintainance Guide" for information about how to modify, extend and maintain the TROL library.

1.1 The Interface of the Robot System

All requests to the A465 robot are processed by the C500 robot controller which is equpipped with an Intel 286 CPU. The C500 provides 2 serial communication channels, one of which is connected to a special input device, the so called teach pendant. The other serial channel is connected to a DOS or Windows 3.1/95/NT PC; normally, this connection is used for character based RAPL command transfer; the controller does, however, also support a proprietary master-slave protocol called **ACI** (**A**dvanced **C**ommunication **I**nterface) which can be used for binary data transfer (e.g. download of RAPL programs into the controller memory).

Furthermore, the controller provides several digital input and output lines. This **GPIO** (**G**eneral **P**urpose **I/O**) connector is located at the controller rear panel.

1.2 Robot Programming

The A465 is normally programmed in RAPL (**R**obotic **A**utomation **P**rogramming **L**anguage), a BASIC-like interpreter language. It is possible either to interactively issue single commands - whereby the PC acts as a terminal - or to load a complete RAPL programm into the controller memory, where it is executed.

What makes the A465 attractive is the possibility to download 8086 code (which was generated with a standard assembler or C-compiler) into the controller's program buffer and execute it there; in CRS terminologie, these programs are called **PCP**s (**P**rocess **C**ontrol **P**rogram). Most of the low level system routines (the so called **RAPL BIOS**) that initiate and control the actions of the robot arm can - similar to the PC BIOS - be called by a PCP via a software interrupt; thus, almost any desired action of the arm can be initiated by a PCP. However, more complex applications, e.g. from the field of computer vision, can not be implemented as a PCP for several reasons:

- The size of a PCP (program code + data) may not exceed 64K.
- It is not possible to use hardware devices such as frame grabbers, LAN-adapters etc.
- PCPs are executed on an Intel 286 CPU, and the code must not be register-optimized. Thus, the execution speed of a PCP is not sufficient for many applications.

1.3 TROL

TROL is a layered communication protocol designed for the data exchange between a DOS/Windows PC and the C500 robot controller. It provides the PC-programmer with a comfortable, high level **API** (**A**pplication **P**rogram **I**nterface) that gives him access to almost all RAPL BIOS functions. The TROL API is a subset of the RAPL command set, but does in some cases provide more functionality (e.g. the *TROLDepart*() command accepts full 6 DOF depart vectors[1], whereas the RAPL DEPART command does only allow for translations). The parameters of any API function are transferred via a serial communication channel to the controller, where the TROL server (which is implemented as a PCP) decodes them and calls the appropriate subroutine of the RAPL BIOS. The application programmer, however, won't have to concern himself with PCP programming.

A TROL API function call will be handled as follows:

1. First, range and plausibility checks of the function arguments are applied. Then, the function code and the parameters are transferred via serial connection to the controller.
2. At the controller, the PCP server receives the request and decodes the parameters; if appropriate, some additional plausibility and safety checks of the arguments will be applied. Finally, the RAPL BIOS is called and its return value transferred back to the calling API function.
3. The API function receives the BIOS code and returns it to the caller.

It is important to understand that TROL API functions behave strictly synchronously: they will not return before the corresponding RAPL BIOS subfunction has been called (or an error has occured). This does, however, not imply that the requested action (e.g. robot arm movement) has finished on return of the API function!

[1] DOF: degrees of freedom; see chapter 4

Any errors will be reported by API return codes. Furthermore, TROL provides several call back adresses which are called in the event of communication errors, thus allowing the user to write his own error handlers.

Chapter 2

Communication

This chapter describes the communication protocol used by TROL to establish a communication link between application and the PCP serverand will give an overwiew over TROL's layered communication model.

2.1 The TROL Layer Model

The architecture of TROL's communication model follows roughly the suggestions of the OSI[1] standard. Below, the four TROL communication layers and their OSI counterparts are given:

TROL		**OSI**	
Physical Layer	(1)	Physical Layer	(1)
Data Link Layer	(2)	Data Link Layer	(2)
Transaction Layer	(3)	Session Layer	(3)
Application Layer	(4)	Presentation Layer	(6) &
		Application Layer	

[1] OSI: **O**pen **S**ystems **I**nterconnection, a (logic) model for communication networks proposed by ISO.

Chapter 3

Installation

This chapter explains how to install and set up TROL for use with your A465 system. Since TROL comes as a single ZIP-file, the first step will be unzipping the file “trol.zip” (e.g. pkunzip -d trol.zip, if you use the pkzip utility).

3.1 Directory Structure

The TROL root directory contains several subdirectories which will be described below.

3.1.1 The TROL Directory Tree

BIN Contains on subdirectory for each supported operating system (e.g. DOS, WIN16 and WIN32). Each of these subdirectories will contain the executables interp.exe (command line interpreter), sinterp.exe (small script interpreter) and cluptrol.exe (allows to terminate the PCP server) for the respective operating system.

DOC Contains documentation in PostScript format.

INSTALL Contains the files pcptrol.exe (PCP server), main.rob (RAPL program needed to activate the PCP server) and trol.cfg (standard TROL configuration file).

EXAMPLE Contains several subdirectories with programming examples. Each of these subdirectories contains a read.me file which contains additional information on how to build and use the examples.

LIB Holds the TROL link libraries for all supported platforms. If more than one compiler is supported for a given platform, there may be more than one library file in some subdirectories (see section 3.1.2 below).

3.1.2 Naming Conventions for Directories and Libraries

Whenever a TROL application can be build for several different operating system / with several different compilers, the names of the directories holding the project files are named as follows: up to 4 characters identify the application (e.g. exp1), three characters identify the operating system (W32 .. 32bit Windows, W16 .. 16bit Windows, DOS .. DOS) and the last character specifies the compiler (B .. Borland, W .. Watcom). The TROL link libraries are also named according to this convention (e.g. trolw32b.lib is the 32bit Windows library for use with Borland C).

3.2 Downloading the PCP Server

We assume that you have either the DOS or Windows version of the CRS utility installed and are familiar with it. From the directory TROL/INSTALL, copy the code of the PCP server and the main RAPL program to the directory in which your CRS utility has been installed. Then start the CRS utility and choose FILE/PCP LOAD. If pcptrol.exe cannot be downloaded, you have probably not set aside enough reserved memory for PCPs and must repartition your controller memory by using the ALLOC and HIMEM commands (please refer to the CRS programming manual on how to do this.) You should reserve at least 20K (better more) for PCP use.

The task of the main.rob RAPL program is to reset the controller baudrate to the baudrate used by TROL, then to launch the PCP server in controller memory and, after the PCP server has terminated, to restore the original comport settings. The version of main.rob that comes with TROL assumes that the serial port number 1 of the controller (the one the PC is connected to) is normally operating at 38400 baud and TROL will be operating at 19200 baud. If this is not true for your system, you must first edit main.rob! If main.rob reflects your system setup, you can download it into the controller program buffer. It is important not to change the name of this program: only if it is named main.rob it can be actived conveniently from the controller front panel via the "Prog. Run/Pause" button.

3.3 Starting a TROL Application

Switch the *point of control* to the controller front panel (see the CRS documentation or our A465 WEB page on how to do this) and push the "Prog. Run/Pause" button. If it illuminates, main.rob will be launched and activate the PCP server in controller memory. For the rest of this discussion, we assume you are running Windows 95 or NT. To test if TROL is working properly, change to the TROL/BIN/WIN32 directory and call interp. Now the prompt of the TROL command line interpreter (CLI) should show up. You may want to try out some of the commands before you leave the CLI by typing "QUIT". If you did not succeed in starting the interpreter, refer to section "troubleshooting".

3.4 Terminating the PCP server

If you now try to restart the CRS utility, it will complain that there is no connection to the controller[1]. This is because the PCP server is still active in controller memory and has the serial interrupt vector for device 1 set to it's own

[1]The Windows version of the CRS utility will hang in this case, so you must kill it from the task manager menu.

interrupt handler. If you want to use the CRS utility after a TROL session, you must first terminate the PCP server; this is accomplished by the utility cluptrol.exe (also located in TROL/BIN/WIN32). As soon as the PCP server terminates, the calling RAPL program main.rob will also return and the light on the "Prog. Run/Pause" button should vanish.

3.5 The TROL Configuration File

Every TROL application must call *InitTROL*() (see section 5.6) before any TROL API function. One of the tasks of *InitTROL*() is to set up the serial communication between PC and robot controller; the default is a connection via serial port COM2 with 19200 baud. These values can be overwritten if you place a TROL configuration file in the same directory as the executable; the name of this file must always be "trol.cfg". The configuration file comprises at least two lines, whereby the first specifies the baudrate and the second the serial port. A third line can be given that specifies a tool transformation that will be issued automatically by *InitTROL*() (see section 4.1. An example TROL configuration file that contains the default settings used by *InitTROL*() is located in the directory TROL/INSTALL.

3.6 Troubleshooting

TROL has been developed with Borland C (BC). This compiler family supports 'project files', which are an improved version of the well-known make files and provide for a comfortable project management. These files have the suffix ".prj" for BC 3.1 resp. ".ide" for versions BC 4.0+.

Chapter 4

Introduction to Robot Programming

This chapter gives a short introduction to robot programming in general, but will also discuss some issues particular to the A465. Many of the terms and notions given here will be needed in later chapters.

4.1 Coordinate Systems

One of the central issues in robot programming is how to specify the current resp. desired position and orientation of the arm (or, more precisely, of the wrist frame - see below). This is usually done in 6 dimensional **cartesian coordinates** <X, Y, Z, YAW, PITCH, ROLL> which are measured relative to the robot's base coordinate system or **base frame** (see Figure 4.1). The first 3 coordinates give the translational offset, wheras the last 3 coordinates describe the orientation of the wrist frame. Orientation is usually given in Z-Y-X EULER or X-Y-Z fixed angles; if the body is initially aligned with the base frame, we first rotate it by an angle YAW about the Z axis, then by an angle PITCH about the new (rotated) Y axis and finally by an angle ROLL about the new X axis. Alternatively, we could first rotate by ROLL about the X, then by PITCH about the Y and finally by YAW about the Z axis of the (non moving) base frame, which would yield the same result.

Cartesian coordinates describe the position and orientation of the so called **wrist frame**, which is located at the center of the flange on the last arm segment (see Figure 4.2), relative to the base frame. An absolute cartesian motion command with target position <X, Y, Z, YAW, PITCH, ROLL> will move the center of the wrist frame to position <X, Y, Z> and orient the wrist as prescribed by <YAW, PITCH, ROLL>. Note that the base frame is fixed, while the wrist frame is moving w.r.t. the base frame.

Normally a tool (e.g. a gripper) will be mounted on the last arm segment, and we are interested in the position and orientation of the **tool frame** (and not of the wrist frame) w.r.t. to the base frame. If the position and orientation of the tool relative to the wrist frame is known, the robot controller will automatically keep track of this additional transform (see next section *TROLSetToolTransform*()). In CRS terminologie, the center of the tool coordinate system is called

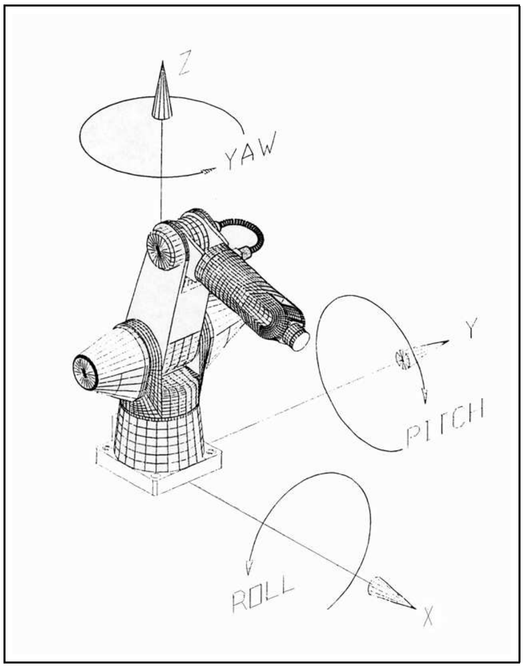

Figure 4.1: The A465 base frame.

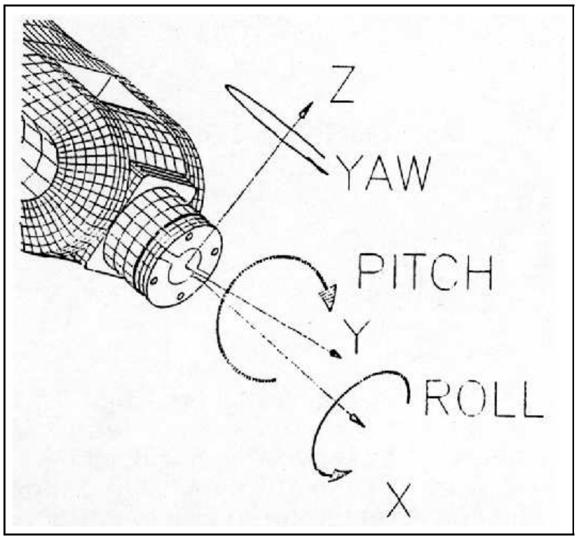

Figure 4.2: The A465 wrist frame.

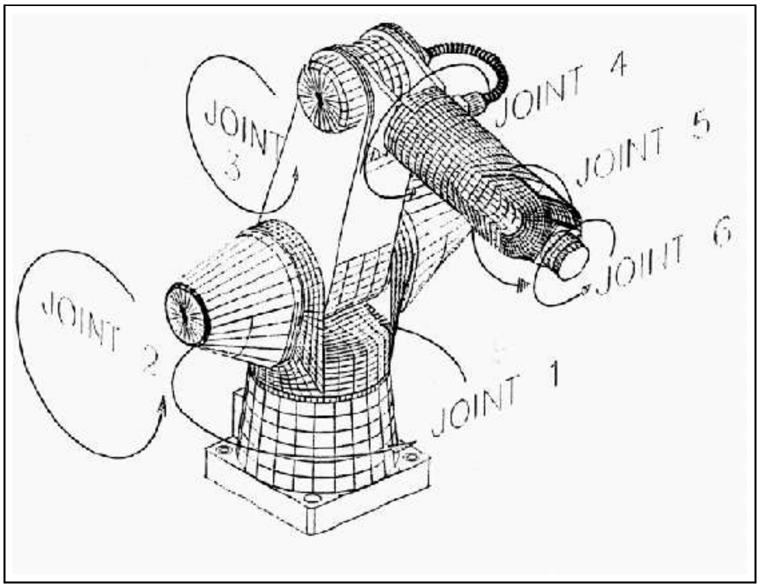

Figure 4.3: The 6 joints of the A465.

the *tool center point* (**TCP**); as long as no tool transformation has been applied, it is coincident with the origin of the wrist frame.

If an application involves several different coordinate systems (frames), there is normally one "special" coordinate system called the **world frame**, relative to which all other frames are known. If the transformation between the robot's base frame and the world frame is known, it is easy to express the position and orientation of the tool in world frame coordinates.

4.2 Forward and Inverse Kinematics

While 6 DOF cartesian coordinates are a device independent and intuitive way to desribe spatial position and orientation, the most 'natural' way to describe the position and orientation of a robot arm are joint angle coordinates, which give the angles beween each pair of neighbouring links of the arm (see Figure 4.3). For any given arm geometry, it is straightforward to calculate the cartesian coordinates of the wrist (or tool) from a given set of joint angles (**forward kinematics**). The inverse problem, namely to determine a set of joint angles which moves the wrist frame to a given position and orientation (**inverse kinematics**), however, is a much harder one. Although for most of modern robot arms there exists a closed form (i.e. non iterative) solution to the inverse kinematics problems, the solution is normally not unique, which means that a given cartesian orientation can be reached with more than one arm configuration (pose).

4.3 Repeatability vs. Accuracy

In typical industrial applications, the robot arm is moved manually (e.g. in limp mode) to the goal positions, where the joint position sensors are read and the current joint angles are stored (teach in). **Repeatability** is a measure of how precisely the arm can return to a taught position; for the A465, this value is 0.05 mm.

3D vision tasks, however, will require the arm to attain **precomputed** 6DOF cartesian postions, possibly positions the arm has never been to before. Before moving the arm, the controller has first to transform cartesian to joint coordinates by applying the inverse kinematics equations of the arm. In the field of robotics, the precision with which such a precomputed position can be attained is referred to as the **accuracy** of the manipulator. Since accurcay does strongly depend on the exact knowledge of the arm's kinematic parameters, it is lower bounded by (and normally much worse than) repeatability.

Chapter 5

TROL API

First, we will introduce some structures and data types which are used with TROL API functions. The following sections describe the API functions themselves. At the start of each section, a list of general return codes for each transaction class is given.

5.1 TROL Data Types

All TROL data types are declared in the header file "troltype.h", which is partly given below:

```
#define POS_PREC  0
#define POS_JOINT 1
#define POS_CART  2

typedef struct
{
    TUCHAR tag;
    TUCHAR numDims;
    union
    {
         TLONG prec[8];
         float joint[8];
         float cart[8];
    }
    data;
} SLocData;
```

Most motion commands require a pointer to a variable of type **SLocData** as an input argument. The union *data* holds - according to the value of *tag* - a set of motor (= precision points), joint or cartesian coordinates. In many cases, most of the coordinates will have 0 value; therefore, in *numDims* you can specify how many coordinates should be transferred to the PCP-server. If, for example, the command *TROLDepart()* should be called with input vector <10, 0, 0, 1.5, 0, 0>, only the first four values have to be sent. If *numdims* contains the value 4, the remaining coordinates will be set to 0 by the PCP-server.

Some of the motion commands accecpt a parameter *straight*; if this parameter has a non-zero value, motion will be done in straight line mode, otherwise in joint interpolated mode.

5.2 API of Transaction Class CAT1

Return Values
These include protocol layer 4, transaction class CAT1 and CAT2 and RAPL BIOS errorcodes. Return values > -32000 are RAPL BIOS errorcodes (which are, however, undocumented). Return values < -32000 are TROL error codes. A return value of 0 does always indicate successful execution of the command.

TINT *TROLAppro*(**TLocName** *locName*, **SLocData** **pDisp*, **TUINT** *straight*)

Moves the TCP to a point whose position and orientation w.r.t. location *locName* is given by *pDisp*. *pDisp* must be specified in 6 DOF cartesian coordinates (*pDisp*$\rightarrow$*tag* == POS_CART) expressed in the tool (not the base!) coordinate system. If the location *locName* does not exist, this function returns the error code TROL_ERR_LOCATION. The following example illustrates how an object situated in a narrow workspace can be approached and gripped by the tool while avoiding collisions.

```
SLocData loc;

loc.tag = POS_CART;
loc.data.cart[0] = -45.0;
loc.numDims = 1;

TROLAppro(``test_001'', &loc, 0);
      /* Approach location test_001 along the */
      /* tool X axis up to 45 mm;             */
      /* during the approach, the tool will   */
      /* assume its final orientation.        */
      /* Use joint interpolated motion (last  */
      /* argument == 0).                      */
TROLMove(``test_001'', 1);
      /* Move the arm to the goal point on a  */
      /* straight line.                       */
TROLFinish(FINISH_WAIT);
      /* Wait 'til motion has finished.       */
TROLClose(60);
      /* Grasp object with 60% force.         */
delay(2000);
      /* Wait 'til gripper is closed.         */
TROLDepart(&loc);
      /* Go cautiously back to the starting   */
      /* position.                            */
```

TINT *TROLApproII*(**SLocData** *pTarget*, **SLocData** **pDisp*, **TUINT** *straight*)

Not yet implemented.

TINT *TROLClose*(**TUINT** *perForce*) Closes the servo gripper with *perForce*% force; *perForce* must be in range [0..100].

WARNING: The servo gripper must not be closed at a force of more than 75% for extended periods (i.e. more than a few seconds)!

TINT *TROLDepart*(**SLocData** **pDisp*, **TUINT** *straight*) Moves the TCP to a point whose position and orientation w.r.t. the current location is given by *pDisp*. *pDisp* must be specified in 6 DOF cartesian coordinates ($pDisp \rightarrow tag$ == POS_CART) expressed in the tool (not the base!) coordinate system.

See also: *TROLAppro*()

TINT *TROLFinish*(**TUINT** *mode*) If the argument *mode* equals FINISH_STATUS, this command will return 0 if there is no motion in progress, 1 otherwise. If FINISH_WAIT is passed as an argument, this function will wait until any ongoing motion has stopped. Since this may take some time, TROL will disable the timeout control while *TROLFinish*() is executing. Note that the RAPL-BIOS will perform an implicit wait between two motion commands, i.e. the second motion command will block until the previous motion has stopped. This may result in timeout errors reported by TROL, so it is usually a good idea to synchronize two consecutive motion commands explicitly using by issuing *TROLFinish*(FINISH_WAIT) after the first one.

Example: see *TROLAppro*()

TINT *TROLGrip*(**TUINT** *dist*) Sets the distance between the two fingers of the gripper to *dist* mm. This command will always work with 100% force. It should thus not be used to grasp an object, but only to position the fingers. *dist* must be in range 0..58 mm (0..2 inches).

TINT *TROLHalt*() Stops immediatly any motion in progress.

TINT *TROLHome*(**TUINT** *axis*) Homing is the process during which the robot arm seeks its so called "homing switches". This is done in order to find the zero marks on the incremental optical encoders associated with each joint axis and thus initialize position sensing. You are allowed to use only incremental motion commands as long as the robot has not been homed, but you should definively home the arm before issuing any motion commands.

axis specifies the axis that shall be homed. Normally, you'll want to home all axes; this can be done by passing an argument value 0.

TINT *TROLJog*(**SLocData** **pDisp*, **TUINT** *straight*) *TROLJog*() is similar to *TROLDepart*() in that it will accept an cartesian ($pDisp \rightarrow tag$ == POS_CART) offset vector as an argument and move the arm by the increment specified in the offset vector; *TROLJog*(), however, will interpret the offset vector in base frame (not tool frame) coordinates. Note also that *pDisp* may contain only a positional offset, which implies that $pDisp \rightarrow numDims$ must be ≤ 3. The *straight* argument specifies, whether the motion should be performed in joint interpolated or straight line mode.

TINT *TROLJoint*(**TUINT** *jointNum*, **float** *radians*) Rotates joint *jointNum* by *radians* radial units; joints are numbered and oriented according to Figure 4.3.

TINT *TROLMA*(**SLocData** **pTarget*) Moves the 6 joints of the A465 to the absolut joint position vector *pTarget*. All angles must be given in radians, whereby *pTarget*→*data.joint*[i-1] holds the new angle for joint i. Clearly, *pTarget* must refer to a joint location (*pTarget*→*tag* == POS_JOINT).

TINT *TROLMI*(**SLocData** **pDisp*) Similar to *TROLMA*(). *pDisp* will, however, be added to the current joint position (incremental move).

TINT *TROLMove*(**TLocName** *locName*, **TUINT** *straight*) Moves the TCP to the controller location *locname*; if the specified location does not exist, TROL_ERR_LOCATION is returned by this function.

TINT *TROLMoveII*(**SLocData** **pTarget*, **TINT** *straight*) Similar to *TROLMove*(); however, the target location is specified directly. Note that *pTarget* must refer to a cartesian or precision point location.

TINT *TROLOnline*(**TUINT** *mode*) This command activates and deactivates the A465 online mode and controls the queue management. The following *mode* flags are accepted: ONLINE_ON, ONLINE_OFF, ONLINE_PROCEED and ONLINE_WAIT. For a more detailed description of the online mode and the meaning of the flags, see the CRS A465 Programming Manual.The ONLINE_WAIT and ONLINE_PROCEED modes are, however, not working as described in the manual (in fact, they seem to do nothing at all!).

TINT *TROLOpen*(**TUINT** *perForce*) Opens the servo gripper with *perForce*% force; *perForce* must be in range [0..100].

WARNING: The servo gripper must not be opened at a force of more than 75% for extended periods (i.e. more than a few seconds)!

TINT *TROLReady*() Moves the arm to the ready position.

TINT *TROLRemote*(**TUINT** *action*, **TUINT** *motionFlag*, **TUINT** *dataFlag*) This command enables (*action* == REMOTE_ON) or disables (*action* == REMOTE_OFF)the A465 remote control mode. In this mode, the arm can be controlled by writing location data directly to an internal memory location of the controller. If you enable remote mode, you have to specify whether the location data specifies relative (*motionFlag* == REMOTE_REL) or absolute (*motionFlag* == REMOTE_ABS) coordinates and whether it will be given in joint (*dataFlag* == REMOTE_JOINT) or cartesian (*dataFlag* == REMOTE_CART) coordinates.

IMPORTANT NOTE: At present, we have no means to control the speed of the arm in remote mode; it will always move at (or near) top speed, so be cautious. If the target position is too far from the current arm position, arm power will fail. Furthermore, CRS has reported that there exist other known problems with this command, but it may take some time to fix them. Furthermore, there is no portable way to determine the address of the internal path buffer, and the address given in the manual may by obsolete or simply wrong (for RAPL version 2.62., it is located at 0x400 in the controller memory). You should therfore not use this command after a firmware update before you have verified that the path buffer address used by the PCP server is still correct!

See also: *TROLRemoteMove*().

TINT *TROLRemoteMove*(**SLocData** **dest*) Writes the location data contained in *dest* to the internal path buffer in the controller. Note that the target location must be given in the coordinate system that was specified as *dataFlag* in the call to *TROLRemote*().

See also: *TROLRemote*().

TINT *TROLSetToolTransform*(**SLocData** **md*) This command informs the controller, where - relative to the original tool coordinate system (i.e. the wrist frame) - the center of a mounted tool is located. *md* must refer to a 6DOF cartesian location. Consult the CRS Programming manual and chapter ??? of this manual for further informations concering tool coordinate system programming.

TINT *TROLSpeed*(**TUINT** *perMaxSpeed*) Sets the speed of arm motion in percent of maximum speed; values for *perMaxSpeed* must be in range [1..100]. Consult the CRS Programming Manual for further details concerning the speed setting for straight line and online motions. This command does not effect the speed of the arm in remote mode.

5.3 API of Transaction Class CAT2

There are currently no CAT2 commands.

5.4 API of Transaction Class CAT3

Return Values
These are the error codes of protocol layer 4.

TINT *TROLGetPosition*(**SLocData** **pos*, **TINT** *mode*) This command returns the current arm position in joint (*pos*→*tag* == POS_JOINT), motor (*pos*→*tag* == POS_PREC) or cartesian (*pos*→*tag* == POS_CART) coordinates. If cartesian coordinates are requested, position and orientation of the tool frame relative to the base frame are retured. The *mode* flag specifies, whether the actual (POS_ACT) or the commanded (POS_COMMANDED) arm position should be returned. The controller will always try to attain and hold the last commanded position; as long as the arm does not move, actual and commanded position are nearly identical (up to repeatability).

Note that a returned cartesian position depends on the actual tool transform setting!

TINT *TROLLimp*(**TUINT** *action*, **TUINT** *axis* Limps *action* == LIMP_ON or unlimps (*action* == LIMP_OFF) joint axis *axis*. If an argument value 0 is passed for *axis*, the specified action will be applied to all joints. While an axis is limped, it can by moved by hand (very useful for teach-in).

This command is not directly supported by the RAPL BIOS and has thus to be passed as a text string to the RAPL interpeter. This means that there are no RAPL BIOS return values (for this reason, *TROLLimp* will always return 0) and any error will crash the PCP server.

5.5 API of Transaction Class CAT4

Return Values
These are the error codes of protocol layer 4.

TINT *TROLTerminatePCP*(**void**) Terminates the PCP server on the robot controller. Should not be called by the user.

TINT *TROLGetStatus*(**TINT** **lastBIOSRetVal*) This function will determine if the last call of a CAT1- or CAT2-transaction has resulted in a call to the RAPL-BIOS; if the BIOS has been called, the last return value of the BIOS is returned in *lastBIOSRetVal*. *TROLGetStatus*() can also be called after the connection to the PCP-server was interrupted; should this happen while a command with *exactly once semanctic* (e.g. an incremental move) is executing, after having reestablished the connection the information provided by *TROLGetStatus*() can be used to decide whether the interrupted command should be issued again or not.

returncode	*lastBIOSRetVal*
0 (BIOS has not been called)	TROL error code
1 (BIOS has been called)	BIOS return code

5.6 Other Functions

In this section we describe some additional functions that are not part of the TROL API, but will be needed to initialize and control serial communication.

TINT *InitTROL*(**TUINT** *serOnly*) This function initializes the serial port given in the configuration file "trol.cfg" and establishes a serial communication link with the PCP server. If *serOnly* is 0, this function will first deactivate the ACI protocol on the controller and then activate the TROL PCP server in controller memory; note that this should normally not be done by the application program itself, but rather by calling the "INITTROL.EXE" utility program. Application programs should call *InitTROL*(1) before any TROL API function.

Possible return values are INITTROL_SUCCESS and INITTROL_FAILURE.

TINT *CleanupTROL*(**void**) Deactivates the PCP server (via *TerminatePCP*()). As long as the PCP server has not been terminated, it is not possible to communicate with the controller in ACI or terminal mode! This function should not be called directly by the user; instead, use the utility program "CLUPTROL.EXE".

Possible return values are INITTROL_SUCCESS and INITTROL_FAILURE.

void *SetL3_TO_Intervall*(**TUINT** *TOIntervall*) After a TROL API function has been called by the user, protocol layer 3 will report a timeout if a receipt is not received within a given time limit. This function sets the timeout intervall (in 1/10 s) of protocol layer 3.

Appendix A

Return Values and Error Codes

A.1 Protocol Layer 4 Error Codes

These codes are returned directly by TROL API functions. While the return code 0 does always indicate successful execution of the command, return codes $<>$ 0 do normally indicate an error condition: codes < -32000 are TROL error codes, wheras codes > -32000 which are returned by any function belonging either to transaction class CAT1 or CAT2 are return codes of the BIOS-subroutine called on the controller. The RAPL BIOS return codes are (at least partially) described in the CRS RAPL Programming Manual. Note that some TROL or BIOS error codes can be filtered out by an user-supplied errorhandler.

A.1.1 General Error Codes

These can be returned by any TROL API function. TROL_ERR_UNKWON, TROL_ERR_BAD_CSIZE and TROL_ERR_BAD_RSIZE typically indicate a version conflict between the PCP server and the TROL link library, so make sure you have installed the most recent versions of both.

-32001 TROL_ERR_COMM A communications error has occurred; the corresponding error handler has been called, but could not fix the problem.

-32002 TROL_ERR_BAD_RSIZE The receipt received from the controller did not have the expected size.

-32003 TROL_ERR_ARG_RANGE One or more of the parameters passed to an API function had invalid values.

-32051 TROL_ERR_UNKWOWN The PCP server did not recognize the commando code.

-32052 TROL_ERR_BAD_CSIZE The query received by the PCP server did not have the expected size.

A.1.2 CAT1 and CAT2 Specific Error Codes

-32101 TROL_RETVAL_RANGE The return value of the called BIOS subfunction is smaller than -32000 (and thus lies in the intervall reserved for TROL error codes). In this case, the return code of the BIOS subfunction can be obtained by calling *TROLGetStatus()*.

-32102 TROL_ERR_LOCATION] The controller location passed as an argument to *TROLMove*() or *TROLAppro*() does not exist.