The CODE Product Family

Customizing CODE

Cimetrix®
INCORPORATED

6979 South High Tech Drive
Salt Lake City, UT 84047-3757
(801) 256-6500  FAX (801) 256-6510
- NOTICE -

This document was written to accompany CODE version 4.0. The information contained in this document is subject to change without notice. Slight variations may occur on computers from varying manufacturers. Every effort has been made to supply complete and accurate information. However, Cimetrix Inc. makes no warranty of any kind with regard to this manual, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Cimetrix shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this manual.

Copyright © 1990 - 1998 Cimetrix Inc.

This document is the property of Cimetrix Inc. and is protected by copyright. The information contained herein is proprietary and may not be copied, transferred, or disclosed to others without prior written authorization from Cimetrix Inc., Salt Lake City, Utah 84047-3757.

Portions of the CODE System are protected by US Patent 4,831,549.

print date: 10/13/98

Trademark Acknowledgments

Terms mentioned in this document that are known to be trademarks are listed below. Cimetrix Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark.

Cimetrix is a registered trademark of Cimetrix, Inc.
The Cimetrix logo, CODE, CIMServer, CIMulation, CIMControl, CIMTools, CODE API and CIMCal are trademarks of Cimetrix Inc.
80386, 80486, and i486 are trademarks of Intel.
PostScript is a registered trademark of Adobe Systems, Inc.
Venture Com
PMAC
MEI
All other products are trademarks or registered trademarks of their respective companies.
# Table of Contents

## CHAPTER 1

- CODE Signal Tables ........................................................................................................... 1-1
  - Using Signal Tables ........................................................................................................ 1-1
  - Creating a Signal Table .................................................................................................. 1-4
  - Signal Table Name Header ............................................................................................. 1-4
  - Driver Definition Block .................................................................................................. 1-4
  - Setting the Driver Type .................................................................................................. 1-4
  - Setting the Driver Instance .............................................................................................. 1-5
  - Defining the Device Name .............................................................................................. 1-5
  - Signal Definition Block .................................................................................................. 1-6
  - Setting a Signal’s Logical Name (Required) ................................................................. 1-6
  - Setting a Signal’s Classification (Optional) ................................................................. 1-7
  - Providing a Description of the Signal (Optional) ......................................................... 1-7
  - Logical Signal Expression (Optional) ............................................................................ 1-7
  - Input Signals, Output Signals, And Initial Values (Required) ........................................ 1-7
  - Signal Parameter Fields ................................................................................................ 1-9
  - CX_SOFTWARE ................................................................................................................. 1-9
  - CX_PMAC_DRV ............................................................................................................... 1-9
  - Defining Simple Signals ................................................................................................ 1-9
  - Defining Complex Signals ............................................................................................. 1-9
    - CX_IPC_DRV .................................................................................................................. 1-12
    - CX_PCL722_DRV ......................................................................................................... 1-14
    - Signal Direction Conflicts ......................................................................................... 1-16
    - CX_AC28_DRV .............................................................................................................. 1-16
    - MEIDSP_DRV .............................................................................................................. 1-17
    - CX_XMP_DRV ............................................................................................................. 1-19
    - CODE XMP Device interface .................................................................................. 1-22
    - XMP Signal Types ..................................................................................................... 1-23
    - FastI0 .......................................................................................................................... 1-23
      - User Memory ........................................................................................................... 1-24
      - Control Memory ...................................................................................................... 1-24
  - Special Comments and Definitions ................................................................................ 1-25
  - Creating the Signal Header File ..................................................................................... 1-26
  - makeheader Usage ......................................................................................................... 1-26
  - PMAC to OAC Conversion .............................................................................................. 2-5
    - Generate a PMAC Data File ..................................................................................... 2-5
    - Keep only the I-Variable and Motor Definitions from the PMAC Data File ................ 2-5
    - Add Card and Motor Definitions ............................................................................. 2-5
    - Add Mechanism Definitions ..................................................................................... 2-5
    - Remove Unused Variables ....................................................................................... 2-7
    - Using the 'Other' Section ......................................................................................... 2-7

## CHAPTER 2

- Mechanism Motor Homing (MMH) ................................................................................... 2-1
  - MMHUtilIPC Generation ............................................................................................... 2-1
  - MMHUtilIPC Menu Items ............................................................................................. 2-2
  - MMHUtilDSP Generation ............................................................................................ 2-2
  - MMHUtilDSP Menu Items .......................................................................................... 2-3
CHAPTER 5 ................................................................................................................................. 5-1

Custom Motion Interfaces .......................................................................................................... 5-1
Introduction .................................................................................................................................. 5-1
CODE Trajectory Control ........................................................................................................... 5-2
Trajectory Generator .................................................................................................................... 5-2
Motion Blending .............................................................................................................................. 5-3
Joint Motion (CX_JOINT_INTERP) .................................................................................................. 5-4
Cartesian Path Motion .................................................................................................................... 5-5
Maintaining Speeds Along Contiguous Path Segments ................................................................. 5-7
Additional Motion Concepts ......................................................................................................... 5-7
Tool Motion Types ........................................................................................................................ 5-7
CX_FULL_POSE ............................................................................................................................. 5-8
CX_Z_POSE and CX_Z_POSE_NO_SPIN ....................................................................................... 5-8
CX_FIXED_ORIENT ....................................................................................................................... 5-9
Tool Control Frame (TCF) .............................................................................................................. 5-9
CHAPTER 6 .................................................................................................................. 6-1

VecTool Interfaces ............................................................................................................. 6-1

Introduction ....................................................................................................................... 6-1

Basic Requirements and Limitations .................................................................................. 6-2

VecTool API’s ..................................................................................................................... 6-3

Implementation ................................................................................................................... 6-4

Add Custom File to Makefile ............................................................................................. 6-4

Add Custom Routine to user_motion VT ........................................................................... 6-5

Compile and Link VecTool Routine .................................................................................... 6-5

New Data Structure Members ............................................................................................. 6-5

Functional Procedures ....................................................................................................... 6-5

Model-Based Procedures ................................................................................................... 6-6

Dynamic Procedures .......................................................................................................... 6-7

A VecTool Example ............................................................................................................. 6-8
Introduction

About this Manual
The material in this manual provides the information you need to maintain, and modify CODE. This manual is for “expert” CODE users that want to modify CODE’s kinematics or create device interfaces with CODE. This manual assumes you are familiar and proficient with CODE as well as the UNIX environment.

How to Use this Manual
In this manual you will learn how to modify CODE to meet your own kinematics and hardware interfacing needs.

See: For:

Chapter 1 A discussion and instruction on how to modify CODE’s signal table.
Chapter 2 A discussion and instruction on how to modify CODE’s kinematics routines. You will learn how to create, compile and link your custom kinematics routines for use with the CIMServer.
Chapter 3 A discussion on the device interfacing architecture of CODE and instruction on how to develop device interfaces to communicate with proprietary hardware devices.
Chapter 4 A complete discussion on CODE motion control. This chapter outlines the interfaces provided the user to control the motion of various mechanisms.
Conventions

The following table lists the meaning or intent in the documentation of various typographical changes such as bold text or different fonts. The functions and keystrokes in the Convention column are given here as examples only.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>typewriter text</td>
<td>Standard UNIX commands, C function names and parameters, and tree node names.</td>
</tr>
<tr>
<td>&lt;Ctrl&gt; o</td>
<td>Press and hold down the Control key while typing the o.</td>
</tr>
<tr>
<td>&lt;Ctrl-Shift&gt; o</td>
<td>Press and hold down the Control and Shift key while typing the o.</td>
</tr>
<tr>
<td>(Optional)</td>
<td>Step is optional, or only applies in certain conditions.</td>
</tr>
<tr>
<td>bold text</td>
<td>Titles of graphical controls (buttons, fields, option and pulldown menus).</td>
</tr>
<tr>
<td>italic text</td>
<td>A newly introduced word or concept which is being defined.</td>
</tr>
</tbody>
</table>

The examples that follow are different formatting types you will see in the documentation:

1. This is an example of a numbered steps in general procedures.

   a. This is a lettered step which represents a substep (a part of performing a single numbered step).

   E1: This is the numbering scheme for specific examples found in the documentation. Examples appear in the documentation to give more specific instances of more general procedures.

   NOTE: This is an example of a note. Notes explain information incidental to the task discussed.

   IMPORTANT: This is an example of an “Important” note. Notes labelled “Important” contain information that affects the functionality of the task discussed.

   WARNING!: This is an example of a “Warning” comment. Warning comments contain information to help you avoid situations that may corrupt data, damage hardware, or harm personnel.
Chapter 1

CODE Signal Tables

The I/O interface of CODE has been implemented using an open architecture, allowing users to implement custom I/O device drivers. The I/O subsystem is used to interface simple devices such as discrete I/O, analog-to-digital converters, and digital-to-analog converters. It can also be used as a signaling mechanism for synchronization of multiple CODE application programs.

CODE uses a signal table to define how each logical signal maps into the physical hardware. This chapter describes how to configure a CODE signal table. For information on implementing a custom I/O device driver, refer to Chapter 0, “Mechanism & I/O Interfacing” in this manual.

Using Signal Tables

Before any signals can be used with CODE software, the signals must be defined in a file called a signal table. When the -sigtable option is used in the cimulation or cimcontrol command line, the signal table is read in by the CIMServer. If the signal table is in a location other than the current directory, then you must include the full path name to the file. If the signal table contains an error, the CIMServer will fail to start. You must correct the errors before you may use the signal table with the Server.

The CIMServer manages the values of all signals declared in the signal table. Any connecting CODE application processes may interact with the defined signals using the API functions defined in the "Events, States, and I/O" in the Programmers Reference Manual. You may not reference any signals in your CODE application process which are not defined in the signal table.

Sample signal tables are included with your copy of the CIMServer (version 3.6.0 or newer). It is recommended that these templates be used for setting up your own signal table. They are located in the following directory:

$ROBTOP/lib/cimetrix/sigtables

NOTE: ROBTOP is an environment variable defining the directory where CODE was installed. Currently, many I/O signal drivers are supported in the CIMServer. These include:

• CX_SOFTWARE  Signals not associated with a hardware I/O interface. These signals are generally used for process coordination and synchronization.

• CX_PMAC_DRV R  Signals associated with the Delta Tau PMAC motion control card.

• CX_IPC_DRV R  Signals associated with the Anorad IPC-2000 motion control card.

• CX_PCL722_DRV R  Signals associated with the Advantech PCL 722 I/O card.
• **CX_AC28_DRVR**  Signals associated with the Opto-22 AC28 I/O card (PAMUX).
• **MEIDSP_DRVR**  Signals associated with a Motion Engineering Incorporated DSP servo controller card.
• **CX_XMP_DRVR**  Signals associated with a Motion Engineering Incorporated XMP servo controller card.

The sample signal file `default_sigtable` contains 10 CX_SOFTWARE signals, the standard CX_PMAC_DRVR 16 digital input, 16 digital output signals, and some signals associated with the Cimetrix Hardware Teach Pendant. The `extended_sigtable` file defines the same signals as those defined in the `default_sigtable` plus signals for accessing PMAC memory and the CX_PMAC_DRVR extended I/O board signals. Sample signal tables for other I/O drivers can be found in the `sigtables` directory.

**NOTE:** In simulation, all signals are considered software signals.

The following is an excerpt from these files (large sections have been skipped):

```plaintext
#---------------CODE Signal Table---------------
Signal table name: sigdefs
#--------------------------------------------------
# SOFTWARE Driver and Signal Definitions
Driver type: 0
Driver instance: 0
NT device name:

Logical signal name: SIGNAL_1
  input  output  use_init  init_value
   1      1       0        0

  field1  field2  field3  field4  field5  field6  field7  field8  field9  field10
  0       0       0       0       0       0       0       0       0       0

Logical signal name: SIGNAL_2
  input  output  use_init  init_value
   1      1       1        1

  field1  field2  field3  field4  field5  field6  field7  field8  field9  field10
  0       0       0       0       0       0       0       0       0       0

# PMAC_DRVR Driver and Signal Definitions
Driver type: 1
Driver instance: 0
#NT device name:

Logical signal name: DI_1
  input  output
   1      0

  init_value
   0
```

1-2  
Customizing CODE
Figure 1-1: Sample Signal Table.
Creating a Signal Table

The CODE signal table is organized into three main sections:
- Signal table name header
- Driver definition blocks
- Signal definition blocks

The signal table name header contains the name of the signal table. The driver definition block contains information specific to the I/O device driver. Signals associated with a specific driver are defined in signal definition blocks.

NOTE: All blank lines and lines beginning with a # (comment line) in a signal table are ignored.

Signal Table Name Header

First, you must give the signal table a name. The first non-comment line of the signal table contains the name of the signal table. This name follows after the colon (:) in the line.

The signal table name is used when you generate a header file for your CODE application processes, as discussed in the section called "Creating the Signal Header File" on page 1-26. For example, if your signal table name was defined:

Signal table name: sigdefs

then the header file sigdefs.h would be created by the header file generator.

Driver Definition Block

A driver definition block defines the driver type, the driver instance, and a logical device name. All signals defined after a driver header become associated with that driver type. Before you may define any signals, you must define a driver header like the one below:

# SOFTWARE Driver and Signal Definitions
Driver type: 0
Driver instance: 0
NT device name:

Setting the Driver Type

The CIMServer uses the Driver Type field to identify a set of specific functions to call to interact with a specific I/O device driver. The driver type constants are defined in $ROBTOP/include/code/cntr_const.h. The supported driver types are defined as follows:

<table>
<thead>
<tr>
<th>Driver</th>
<th>Driver Type</th>
<th>NT Supported</th>
<th>RTX Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>CXSOFTWARE</td>
<td>0</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CXPMADEVR</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CXIPCDEVR</td>
<td>3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CXPCL722DEVR</td>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CXCAC28DEVR</td>
<td>6</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
When you enter the constant in the field for the driver type, you must conform to the following format:

**Driver Type: 0**

If you have defined a custom device driver, then you would specify the number associated with that device driver.

**IMPORTANT:** If the CIMServer is running in simulation mode (server started with `simulation` command line option), then all signals are treated as if they are **CX_SOFTWARE** driver types.

### Setting the Driver Instance

For driver types that are used more than once in the same signal table (e.g. multiple I/O cards of the same type in a single bus), you must set the number of the instance of that particular type of driver. For cases where a driver type is used in only one driver header, you must enter a 0 (zero) in this field as follows:

**Driver Instance: 0**

If more than one signal definition block exists for the same driver type, then the driver instance number should increment from 0 for each signal definition block.

### Defining the Device Name

The interpretation of this field is dependent on the implementation of the custom driver type. For example, if a device communicates with the CPU host through a serial port, the logical name of a serial communication port might be used as the device name.

**CODE** supports only NT device drivers. In other words, you must use the following line to define NT device name:

**NT device name:**

Assigning the device name strictly depends on the driver’s implementation. Even if the device name is ignored by the driver, you must still include the device name label and colon (:) for the signal table to be read correctly by the CIMServer. When the driver ignores the device name, you may choose to not enter a name, as shown below:

**NT device name:**

The device name field is defined as follows for the supported driver types:

<table>
<thead>
<tr>
<th>Driver Type</th>
<th>Device Name Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_SOFTWARE</td>
<td>The device name is ignored by the CX_SOFTWARE driver.</td>
</tr>
<tr>
<td>CX_PMAC_DRVR</td>
<td>The device name is ignored by the CX_PMAC_DRVR driver.</td>
</tr>
<tr>
<td>Driver Type</td>
<td>Device Name Format</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>CX_IPC_DRVR</td>
<td>First card: /dev/ipc0</td>
</tr>
<tr>
<td></td>
<td>Second card: /dev/ipcl</td>
</tr>
<tr>
<td>CX_PCL722_DRVR</td>
<td>First card: UNIX /dev/pcl722_0 NT ./pcl722_0</td>
</tr>
<tr>
<td></td>
<td>Second card: UNIX /dev/pcl722_1 NT ./pcl722_1</td>
</tr>
<tr>
<td>CX_AC28_DRVR</td>
<td>First card: /dev/ac28_0</td>
</tr>
<tr>
<td></td>
<td>Second card: /dev/ac28_1</td>
</tr>
<tr>
<td>MEIDSP_DRVR</td>
<td>The device name is ignored by the MEIDSP_DRVR driver.</td>
</tr>
<tr>
<td>CX_XMP_DRVR</td>
<td>The device name is ignored by the CX_XMP_DRVR driver.</td>
</tr>
</tbody>
</table>

### Signal Definition Block

Once the driver definition block is complete, you may define signals associated with that driver type. Each signal definition contains up to eight lines of descriptive fields as follows:

- **Logical signal name:** SIGNAL_1
- **Class:** Signal Classification string (optional)
- **Expression:** combination signal (optional)
- **Description:** Signal description string (optional)

```
input output use_init init_value
1 1 0 0
field1 field2 field3 field4 field5 field6 field7 field8 field9 field10
0 0 0 0 0 0 0 0 0 0
```

The sections that follow explain the descriptive fields.

**NOTE:** The Class, Expression, and Description fields are optional. The other fields (labels and values) must be included in each signal definition, even when some of the fields 1 through 10 are ignored by the driver.

### Setting a Signal’s Logical Name (Required)

The CIMServer automatically assigns each signal a number based on the signal definition order, starting from 0 (zero). In CODE application processes, signals are referenced using this numerical index into the signal table. However, for code readability purposes, you may wish to use a more intuitive logical name rather than the numerical index into the signal table when referencing I/O signals in an application.

CODE includes a utility (makeheader) which automatically generates a header file in which the logical name is mapped (using #define's) into the corresponding numerical index into the signal table. This utility is described later in the "Creating the Signal Header File" on page 1-26.

The signal logical name is only used when generating a header file for CODE application processes using the makeheader utility. As an example, if you defined the following signal logical names:

```
Logical signal name: BLUE_LIGHT
input output use_init init_value
1 1 0 0
field1 field2 field3 field4 field5 field6 field7 field8 field9 field10
0 0 0 0 0 0 0 0 0 0
```
Logical signal name: PALLET READY

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>field1</th>
<th>field2</th>
<th>field3</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

then the following definitions would be generated in your header file, assuming that these were the first signals defined in the signal table:

```c
#define BLUE_LIGHT 0
#define PALLET_READY 1
```

If you included the header file in your CODE application process, you could specify `BLUE_LIGHT` instead of 0 when using signal API functions.

You should give each signal a meaningful logical name. Each signal must have a unique logical name with a maximum of 50 characters.

**Setting a Signal’s Classification (Optional)**

The Class field allows logical I/O signals to be grouped by class. This field is a character string of up to 256 characters. A signal’s class can be determined using the `CxGetSignalClass()` CODE API function.

**Providing a Description of the Signal (Optional)**

The Description field is read by the CIMServer when the signal table is loaded. The Description is a character string which can be up to 256 characters in length. The description can be obtained in a CODE application program using the `CxGetSignalDescription()` CODE API function.

**Logical Signal Expression (Optional)**

This field allows you to specify some logical signal combinations of the signals defined in the signal table. This field has meaning only if the specified device driver supports such functionality.

**Input Signals, Output Signals, And Initial Values (Required)**

Once signal values are read into the CIMServer, the Server keeps the signal values in an internal table. When the values change, the Server reacts appropriately. The following fields define how the CIMServer interacts with the driver and CODE application processes:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

These fields have the same meaning, regardless of the driver type, and are defined in the table below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>This determines whether the device driver will be used to query the signal value. You must specify the input type as 0, 1, or 2.</td>
</tr>
<tr>
<td>Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>The CIMServer gets the value from its internal table without querying the driver.</td>
</tr>
<tr>
<td>1</td>
<td>The CIMServer polls the value from the driver, and saves the value in its internal table. This signal must be polled by the CIMServer to determine if the signal has changed.</td>
</tr>
<tr>
<td>2</td>
<td>The CIMServer is interrupted by the driver when the value changes, and the new value is saved in its internal table.</td>
</tr>
</tbody>
</table>

**NOTES:**
- This field is must be set to 1 for CX_SOFTWARE signals.
- If a physical signal is strictly an output, then the input type must be 0.
- If a physical signal is strictly an input, then the input type must be 1 (e.g. a detection device).
- Input signals may only be read.

<table>
<thead>
<tr>
<th>output</th>
<th>This determines whether a CODE application process may set the signal’s value. You must specify the output type as 0 or 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CIMServer does not permit a CODE application process to set the signal value. An error is returned if a CODE application process tries to set the signals value. This setting is used for input signals.</td>
</tr>
<tr>
<td>1</td>
<td>CIMServer saves the value in its internal table, and writes the value to the driver. This setting is used for output signals.</td>
</tr>
</tbody>
</table>

**NOTES:**
- This field is must be set to 1 for CX_SOFTWARE signals.
- If a physical signal is strictly input, then the output type must be 0.
- If a physical signal is strictly output, then the output type must be 1 (e.g. an On/Off switch).
- If the CIMServer is running in simulation mode, and the use_init flag is set to 0, the output is always initialized to 0.
- Output signals may be read or set.

<table>
<thead>
<tr>
<th>use_init</th>
<th>Determines whether the signal is initialized to init_value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CIMServer does not initialize the value on start up.</td>
</tr>
<tr>
<td>1</td>
<td>CIMServer sets the signal to the value in init_value on start up.</td>
</tr>
</tbody>
</table>

**NOTES:**
- If the output field is 0, the use_init field must also be 0.

| init_value | If use_init equals 1, then the CIMServer initializes the signal to this value. |
Signal Parameter Fields

After setting the `input`, `output`, `use_init`, and `init_value` fields, you must define the signal’s parameters. These parameters define the logical to physical mapping for the specific hardware being used. The signal table incorporates ten driver specific labels and fields for this purpose. The labels are included for descriptive purposes only, and are read but not used by the Server. The interpretation of the corresponding data fields are unique for each driver type. The fields for each supported driver are defined as follows:

**CX_SOFTWARE**

For `CX_SOFTWARE` signals, these fields are ignored, so they should all be set to 0 (ZERO).

**CX_PMAC_DRVR**

For `CX_PMAC_DRVR` signals, the first field, labeled `sigtype` for convenience, determines whether a signal is simple(0) or complex(1). The nine fields which follow are interpreted differently, depending on the signal type. These fields are explained in the following sections.

Important: You must enter values in all ten fields whether they are all used or not, or the signal table will not be read correctly by the CIMServer.

**Defining Simple Signals**

If the `sigtype` field of a `CX_PMAC_DRVR` signal is 0 (zero), it becomes a simple signal. Simple signals are the standard 16 digital input signals and 16 digital output signals. Simple signals only require you to enter a number under the label `pnum`, the second field. This field is defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pnum</code></td>
<td>Number used by the <code>CX_PMAC_DRVR</code> device driver to identify physical signal (numbers 1-16 for input, and 1-16 for output signals). This field is used only if <code>sigtype</code> is 0</td>
</tr>
</tbody>
</table>

The following example shows how a simple signal’s parameters are defined:

```
0 1 0 0 0 0 0 0 0 0
```

**Defining Complex Signals**

If the first field of a `CX_PMAC_DRVR` signal is 1 (one), it becomes a complex signal. Any PMAC card memory location can be used as a complex signal. If you are defining complex signals, you must enter values for the different memory locations on the PMAC card.
IMPORTANT: This section assumes you are familiar with the PMAC card. For more information on the PMAC card, see the PMAC User’s Manual and Software Reference.

For complex signals you must fill in the fields `mtype, byte, bit, width, and sign` as shown in the following example:

```
sigtype  pnum  mtype  byte  bit  width  sign  field8  field9  field10
   1  0  1  0xd10  0  24  0  0  0  0
```

The following is an excerpt from a signal table which defines some PMAC signals:

```c
# PMAC_DRVR Driver and Signal Definitions
Driver type: 1
Driver instance: 0
Device name:

#Outputs
Logical signal name: HIGH_POWER_ENABLE
```

```c
        input  output  use_init  init_value
          0      1      0      0

sigtype  pnum  mtype  byte  bit  width  sign  field8  field9  field10
   1  0  2  0xC080  1  1  0  0  0  0  0

Logical signal name: SERVO_ON_BYPASS
```

```c
        input  output  use_init  init_value
          0      1      0      0

sigtype  pnum  mtype  byte  bit  width  sign  field8  field9  field10
   1  0  2  0xC080  2  1  0  0  0  0  0

Logical signal name: SET_ALARM_LAMP
```

```c
        input  output  use_init  init_value
          0      1      0      0

sigtype  pnum  mtype  byte  bit  width  sign  field8  field9  field10
   1  0  2  0xC080  3  1  0  0  0  0  0

#INPUTS
Logical signal name: BRAKE_1_CLAMPED
```

```c
        input  output  use_init  init_value
          2      0      0      0

sigtype  pnum  mtype  byte  bit  width  sign  field8  field9  field10
   1  0  2  0xC080  8  1  0  0  0  0  0
```
Logical signal name: SET_ALARM_LAMP

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sigtype</th>
<th>pnum</th>
<th>mtype</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>sign</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0xC080</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: SET_ALARM_LAMP

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sigtype</th>
<th>pnum</th>
<th>mtype</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>sign</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0xC080</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The parameter fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pnum</td>
<td>Ignored (set to 0)</td>
</tr>
<tr>
<td>mtype</td>
<td>Memory type. Possible entries are described below:</td>
</tr>
<tr>
<td></td>
<td>0  ‘*’ generic mvar location (no byte, bit, width, or sign)</td>
</tr>
<tr>
<td></td>
<td>1  ‘X’ 24 bit word (uses byte, bit, width, and sign)</td>
</tr>
<tr>
<td></td>
<td>2  ‘Y’ 24 bit word (uses byte, bit, width, and sign)</td>
</tr>
<tr>
<td></td>
<td>3  ‘D’ 48 bit fixed point (uses byte only)</td>
</tr>
<tr>
<td></td>
<td>4  ‘L’ 48 bit floating point (uses byte only)</td>
</tr>
<tr>
<td></td>
<td>5  ‘DP’ 32 bit DPRAM fixed point (uses byte only)</td>
</tr>
<tr>
<td></td>
<td>6  ‘F’ 32 bit DPRAM floating point (uses byte only)</td>
</tr>
<tr>
<td></td>
<td>7  ‘TWB’ binary thumbwheel (uses byte, bit, width, and sign (0, 1, 2))</td>
</tr>
<tr>
<td></td>
<td>8  ‘TWD’ BCD thumbwheel (uses byte, bit, width, and sign (0, 1, 2))</td>
</tr>
<tr>
<td></td>
<td>9  ‘TWS’ serial thumbwheel (uses byte only)</td>
</tr>
<tr>
<td>byte</td>
<td>Offset address from signal base address identifying where the MSB of the signal is located. Possible range of values are: 0x0 – 0xFFF</td>
</tr>
<tr>
<td>Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>bit</td>
<td>The starting bit within the byte. Possible range of values are: 0 - 23. The bit parameter values are dependent on the width parameter, when the width parameter is required as defined below:</td>
</tr>
<tr>
<td>width</td>
<td>possible bit values</td>
</tr>
<tr>
<td>1</td>
<td>0 - 23</td>
</tr>
<tr>
<td>4</td>
<td>0, 4, 8, 12, 16, 20</td>
</tr>
<tr>
<td>8</td>
<td>0, 4, 8, 12, 16</td>
</tr>
<tr>
<td>12</td>
<td>0, 4, 8, 12</td>
</tr>
<tr>
<td>16</td>
<td>0, 4, 8</td>
</tr>
<tr>
<td>20</td>
<td>0, 4</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>width</td>
<td>Bit width of the signal from the base address + byte address. Possible values include: 1, 4, 8, 12, 16, 20, 24.</td>
</tr>
<tr>
<td>sign</td>
<td>Signed flag. Possible values are described below.</td>
</tr>
<tr>
<td>0</td>
<td>No sign specified</td>
</tr>
<tr>
<td>1</td>
<td>‘U’ unsigned</td>
</tr>
<tr>
<td>2</td>
<td>‘S’ signed</td>
</tr>
<tr>
<td>3</td>
<td>‘D’ binary coded decimal (BCD)</td>
</tr>
<tr>
<td>4</td>
<td>‘C’ complimentary coded decimal (CCD)</td>
</tr>
</tbody>
</table>

**CX_IPC_DVR**

The following is an excerpt from a signal table which defines some PMAC signals:

```plaintext
# IPC_DVR Driver and Signal Definitions
Driver type: 3
Driver instance: 0
Device name: /dev/ipc0
NT Device name: \IPC2000_0

# INPUTS
Logical signal name: DI_1

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype pnu mtype byte bit width sign field8 field9 field10
```

1-12  
*Customizing CODE*
m

1 8 0 0 0 0 0 0 0 0 0 0

Logical signal name: DI_2
input output use_init init_value
1 0 0 0

sigtype pnum mtype byte bit width sign field8 field9 field10
1 8 0 0 0 0 0 0 0 0 0

#OUTPUTS
Logical signal name: D0_1
input output use_init init_value
1 1 0 0

sigtype pnum mtype byte bit width sign field8 field9 field10
1 0 0 0 0 0 0 0 0 0 0

Logical signal name: D0_2
input output use_init init_value
1 1 0 0

sigtype pnum mtype byte bit width sign field8 field9 field10
1 1 0 0 0 0 0 0 0 0 0

The following example shows how an IPC-2000's signal parameters are defined.

sigtype pnum mtype byte bit width sign field8 field9 field10
1 1 0 0 0 0 0 0 0 0 0

The parameter fields are defined as follows:

<table>
<thead>
<tr>
<th>field</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigtype 0</td>
<td>1-bit digital signal 1-8. This setting is used when the signal position will be indicated by pnum.</td>
</tr>
<tr>
<td>1</td>
<td>1-8 bit digital word 0-255. This setting is used when a bitmask indicates signal position.</td>
</tr>
</tbody>
</table>
### field explanation

<table>
<thead>
<tr>
<th>field</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pnum</td>
<td>This field determines the signal position number. For signals of &quot;sigtype 0&quot; this field defines the signal number 1 through 8. For signals of &quot;sigtype 1&quot; this field defines a mask of bits (e.g. a mask of 5 means that inputs/outputs 1 and 4 will be set or received.</td>
</tr>
</tbody>
</table>

## CX_PCL722_DRVR

The following is an excerpt from a signal table which defines some PCL-722 signals. This signal table defines four signals associated with PCL-722 device /dev/pcl722_0. This first signal is an 8-bit input signal on port A of connector 0 called PCL_IN_0_A. The second signal is an 8-bit output signal on port B of connector 0 called PCL_OUT_0_B. The third signal is a 4-bit input signal on the least significant half of port C on connector 0. The fourth signal is a 1-bit output signal on one bit of the most significant half of port C on connector 0.

### CODE Signal Table

Signal table name: sigdefs_pcl

```bash
#--------------------CODE Signal Table--------------------
Signal table name: sigdefs_pcl

# PCL722_DRV Driver and Signal Definitions
Driver type: 5
Driver instance: 0
Device name: /dev/pcl722_0
NT device name: \.\pcl722_0

Logical signal name: PCL_IN_0_A
<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conn</th>
<th>port</th>
<th>width</th>
<th>bit</th>
<th>invert</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: PCL_OUT_0_B
<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conn</th>
<th>port</th>
<th>width</th>
<th>bit</th>
<th>invert</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: PCL_IN_0_CLS
<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conn</th>
<th>port</th>
<th>width</th>
<th>bit</th>
<th>invert</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: PCL_OUT_0_C4
```
<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>The field indicates the number of the connector on the PCL-722 associated with the signal. Valid connector numbers are 0 through 5.</td>
</tr>
<tr>
<td>port</td>
<td>The field indicates the port associated with the signal. A value of 0 indicates Port A. A value of 1 indicates Port B. A value of 2 indicates port C.</td>
</tr>
<tr>
<td>width</td>
<td>The number of bits contained in the signal. For the PCL-722 valid widths are:</td>
</tr>
<tr>
<td>8</td>
<td>byte wide signal</td>
</tr>
<tr>
<td>4</td>
<td>4-bit wide signal</td>
</tr>
<tr>
<td>1</td>
<td>single bit signal</td>
</tr>
<tr>
<td>bit</td>
<td>The starting bit within the word. The bit parameter values are dependent on the width of the signal:</td>
</tr>
<tr>
<td>1-bit signal</td>
<td>bit can be: 0-7 (0 = LS bit)</td>
</tr>
<tr>
<td>4-bit signal</td>
<td>bit can be: 0 = LS nibble, 4 = MS nibble</td>
</tr>
<tr>
<td>8-bit signal</td>
<td>0 is only legal bit value</td>
</tr>
<tr>
<td>invert</td>
<td>Invert the bit-sense of the signal (i.e. all 1 bits become 0, and all 0 bits become 1).</td>
</tr>
</tbody>
</table>
Signal Direction Conflicts

On the PCL-722 board, all signals defined on a particular port of a connector (or half port in the case of port C), must be the same direction. It is illegal to define an input and an output signal on port A of a connector, for example. CIMServer will abort with an error code of PCL722_PORT_CONFIG_CONFLICT if it detects such an error in the signal table.

CX_AC28_DRVR

The following is an excerpt from a signal table which defines some PAMUX signals connected to an AC28 board. This signal table defines three signals associated with an AC28 device called /dev/ac28_0. The first signal is an 8-bit input signal called PAMUX_IN_0. The second signal is an 8-bit output signal called PAMUX_OUT_0. The third signal is a 16-bit input signal called PAMUX_IN_1.

---

# AC28_DRVR Driver and Signal Definitions
Driver type: 6
Driver instance: 0
Device name: /dev/ac28_0

Logical signal name: PAMUX_IN_0
input output use_init init_value
1 0 0 0

<table>
<thead>
<tr>
<th>brdIO</th>
<th>width</th>
<th>bit</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: PAMUX_OUT_0
input output use_init init_value
0 1 0 0

<table>
<thead>
<tr>
<th>brdIO</th>
<th>width</th>
<th>bit</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: PAMUX_IN_1
input output use_init init_value
1 0 0 0

<table>
<thead>
<tr>
<th>brdIO</th>
<th>width</th>
<th>bit</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Figure 1-4: Sample AC28 signal table.
For the AC28 driver, only the first three fields are used and the others should be set to zero.

IMPORTANT: You must enter values in all ten fields whether they are all used or not, or the signal table will not be read correctly by the CIMServer.

The following example shows how an AC28 PAMUX signal’s parameters are defined:

<table>
<thead>
<tr>
<th>brdIO</th>
<th>width</th>
<th>bit</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

These fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>brdIO</td>
<td>The field indicates the base PAMUX bus address of the I/O bank containing the signal. I/O numbers are in multiples of 4 starting with 0.</td>
</tr>
<tr>
<td>width</td>
<td>This field indicates the width, in bits, of the PAMUX signal. With the AC28 driver, signal widths between 1 and 32 are permitted. However, a signal is not allowed to cross a 32-bit boundary.</td>
</tr>
<tr>
<td>bit</td>
<td>This field indicates the starting bit position of the signal within a 32-bit bank. Bit 0 is the least significant bit.</td>
</tr>
</tbody>
</table>

As indicated above, the CODE AC28 PAMUX driver allows signals to be between 1 and 32 bits in width. However, a signal may not cross a 32-bit boundary.

There is a performance advantage to defining wider signals rather than defining each I/O point as an individual 1-bit signal. The CIMServer can poll a 32-bit wide signal as fast as it can a 1-bit wide signal. Thus, it is about 32 time faster to poll a single 32-bit wide signal than it is to poll 32 individual 1-bit wide signals.

**MEIDSP_DVR**

The following is an excerpt from a signal table which defines some I/O signals associated with an MEI DSP servo card. This signal table defines three signals associated with the MEI card.

#------------------CODE Signal Table------------------
Signal table name: sigdefs_mei
#--------------------------------------------------
# MEIDSP_DRV Driver and Signal Definitions
Driver type: 8
Driver instance: 0
NT device name: \\dspio0

Logical signal name: CONVEYOR_LIFT_UP_SENSOR
input output use_init init_value
1 0 0 0

portNum portAddr portType portCfg bit width invert field8 field9 field10
For the MEI driver, only the first seven fields are used and the others should be set to zero. IMPORTANT: You must enter values in all ten fields whether they are all used or not, or the signal table will not be read correctly by the CIMServer.

The following example shows how an MEI DSP signal’s parameters are defined:

These fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>portNum</td>
<td>Unused. Reserved for future use.</td>
</tr>
<tr>
<td>portAddr</td>
<td>MEI port number</td>
</tr>
<tr>
<td>portType</td>
<td>Port type</td>
</tr>
<tr>
<td>portCfg</td>
<td>Port configuration</td>
</tr>
<tr>
<td>bit</td>
<td>Used for digital I/O signals</td>
</tr>
<tr>
<td>width</td>
<td>Used for Analog I/O signals</td>
</tr>
<tr>
<td>invert</td>
<td>Port configuration</td>
</tr>
<tr>
<td>field8</td>
<td>Digital I/O ports (4-axis cards).</td>
</tr>
<tr>
<td>field9</td>
<td>Digital I/O ports (8-axis cards only).</td>
</tr>
<tr>
<td>field10</td>
<td>Analog ports (8-axis cards only).</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0 - Bitmask</td>
<td>Used with digital I/O ports (0-5). In this case the bit and width fields define the start address and signal width, respectively.</td>
</tr>
<tr>
<td>1 - Logical</td>
<td>Used with digital I/O ports (0-5). In this case the bit and width fields define the AND and OR gates through which the signal value will be passed.</td>
</tr>
<tr>
<td>2 - MinMax</td>
<td>Used with digital I/O ports (0-5). In this case the bit and width fields correspond to min and max signal values.</td>
</tr>
<tr>
<td>1 - Differential</td>
<td>Used with Analog I/O ports (10-17). If bit 0 is set then the signal is differential. Otherwise, it is single-ended. In this case bit and width define the min and max signal values.</td>
</tr>
<tr>
<td>2 - Bipolar</td>
<td>Used with Analog I/O ports (10-17). If bit 1 is set then the signal is Bipolar. Otherwise, it is unipolar. In this case, bit and width define the min and max values.</td>
</tr>
<tr>
<td>3 - Diff &amp; Bipolar</td>
<td>Used with Analog I/O port (10-17). Both Differential and Bipolar I/O signals.</td>
</tr>
<tr>
<td>invert</td>
<td>Invert the bit-sense of the signal (i.e. all 1 bits become 0, and all 0 bits become 1).</td>
</tr>
</tbody>
</table>

**CX_XMP_DRVR**

The following is an example signal table which defines some I/O signals associated with an MEI XMP servo card:

```markdown
# MEI XMP Card Driver and Signal Definitions
Driver type: 17
Driver instance: 0
NT device name:

Logical signal name: FASTIO_INPUT_XCVR_A0

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_INPUT_XCVR_A1

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_INPUT_XCVR_A2

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_OUTPUT_XCVR_A3

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_OUTPUT_XCVR_A4

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_OUTPUT_XCVR_A5

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_INPUT_XCVR_A6

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_INPUT_XCVR_A7

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>XCVR</th>
<th>Bit Width</th>
<th>Invert</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical signal name: FASTIO_INPUT_XCVR_A8

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Use_Init</th>
<th>Init_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Customizing CODE
<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** FASTIO_OUTPUT_XCVR_A9

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** FASTIO_OUTPUT_XCVR_A10

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** FASTIO_OUTPUT_XCVR_A11

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** USER_MEMORY1

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** USER_MEMORY2

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logical signal name:** USER_MEMORY3

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field 8</th>
<th>field 9</th>
<th>field 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical signal name: CONTROL_SIGNATURE

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype  byte  bit  width  invert  field 6  field 7  field 8  field 9  field 10
1  0x0  0  32  0  0  0  0  0  0

Logical signal name: MS0_AXES

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype  byte  bit  width  invert  field 6  field 7  field 8  field 9  field 10
2  0x5aa8  0  32  0  0  0  0  0  0

Logical signal name: MS0_MODE

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype  byte  bit  width  invert  field 6  field 7  field 8  field 9  field 10
2  0x5aac  0  32  0  0  0  0  0  0

Logical signal name: MS0_ACTION

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype  byte  bit  width  invert  field 6  field 7  field 8  field 9  field 10
2  0x5b34  0  32  0  0  0  0  0  0

Logical signal name: MS0_STATUS

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sigtype  byte  bit  width  invert  field 6  field 7  field 8  field 9  field 10
2  0x5b4c  0  32  0  0  0  0  0  0

**CODE XMP Device interface**

There are a number of additional files that have been added to CODE to support the XMP card. These files, along with their standard locations, are as follows:

%ROBTO%in
cimcontrol.exe
XMP Signal Types

There are three different types of signals that can be used with the XMP card. The three types are FastIO, User memory, and Control memory. Each of these signal types is described below along with the associated signal definition block.

FastIO

FastIO signals have physical connections to the real world. There are twelve FastIO signals for every four axes on the XMP card. Each set of four axes has its own transceiver. FastIO signals can be configured to be either inputs or outputs. If the same physical signal is defined in two separate signals definitions blocks to be both an input and an output, it will be configured as an output. The signal definition block for a FastIO signal looks like this:

| Logical signal name: FASTIO_SIGNAL |
|-------------------------------|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| input | output | use_init | init_value |
| 1 0 0 0 |

<table>
<thead>
<tr>
<th>type</th>
<th>xcvr</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The type field must be 0 (zero) for a FastIO signal.

The xcvr field corresponds to the transceiver on which the signal is located, and must be 0-3, inclusive. The following table shows the correct value for the xcvr field for different locations of the FastIO signal:

<table>
<thead>
<tr>
<th>xcvr field</th>
<th>Axes</th>
<th>Board</th>
<th>Transceiver letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-3</td>
<td>Main</td>
<td>A</td>
</tr>
</tbody>
</table>
The bit field indicates the first bit of interest and must be 0-15, inclusive. In normal use, this number corresponds with the transceiver signal number and should be 0-11, inclusive. Bit numbers 12-15 are defined by MEI. Refer to MEI documentation for these bit definitions.

The width field indicates the number of bits contained in the signal and must be 1-16 inclusive. The sum of the width and bit fields must be less than 17.

The invert field indicates whether to invert the value of the signal. This field must be either a 0 or a 1. This field is intended to be used when the sense of the signal is inverted by some external circuitry, such as optical isolation.

**User Memory**

User memory signals do not have physical connections to the real world. They exist only on the XMP card. They are typically used to meet high performance coordination requirements with CxQueuedWaitForSignal, or to communicate with XMP program sequencers. There are currently 252 bytes available. User memory signals can be configured to be either inputs or outputs. The signal definition block for a user memory signal looks like this:

```
Logical signal name: USER_MEMORY
input output use_init init_value
1 1 1 0
```

```
type byte bit width invert field6 field7 field8 field9 field10
1 0 0 32 0 0 0 0 0 0
```

The type field must be 1 (one) for a user memory signal.

The byte field indicates the offset from the start of user memory and must be 0-248, inclusive, and evenly divisible by 4.

The bit field indicates the first bit of interest and must be 0-31, inclusive.

The width field indicates the number of bits to be contained in the signal and must be 1-32 inclusive. The sum of the width and bit fields must be less than 33.

The invert field indicates whether to invert the value of the signal. This field must be either a 0 or a 1. This field is intended to be used when the sense of the signal is inverted by some external circuitry, such as optical isolation.

**Control Memory**

Control memory signals also do not have physical connections to the real world. They exist only on the XMP card. They are typically used to monitor card status, or to communicate with XMP program sequencers. Control memory signals can be configured to be either inputs or outputs, but should typically be used as inputs only. It is possible to cause the card to malfunction if control memory is written to indiscriminately. Control memory is
arranged in a structure of type MEIXmpData, defined in header files supplied by MEI. The signal definition
block for a control memory signal looks like this:

<table>
<thead>
<tr>
<th>Logical signal name: CONTROL_MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>byte</th>
<th>bit</th>
<th>width</th>
<th>invert</th>
<th>field 6</th>
<th>field 7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The type field must be 2 (two) for a control memory signal.

The byte field indicates the offset from the start of the control memory and must be evenly divisible by 4.

The bit field indicates the first bit of interest and must be 0-31, inclusive.

The width field indicates the number of bits contained in the signal and must be 1-32 inclusive. The sum of
the width and bit fields must be less than 33.

The invert field indicates whether to invert the value of the signal. This field must be either a 0 or a 1. This
field is intended to be used when the sense of the signal is inverted by some external circuitry, such as optical
isolation.

**Special Comments and Definitions**

When the signal table is defined, special comments or constants can be defined for use with the
makeheader (see "Creating the Signal Header File" on page 1-26) utility. These comments begin
with the '%' character. When the signal table is read by the CIMServer, these lines are ignored.
However, when the signal table is scanned by the makeheader utility, the '%' symbols are stripped,
and the remaining characters are placed in the resulting header file.

Example:
The following segment of a signal

```%
#define GRIPPER_OPEN 1
#define GRIPPER_CLOSE 0
%
```

Logical signal name: GRIPPER_HAND

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>use_init</th>
<th>init_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>brdIO</th>
<th>width</th>
<th>bit</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th>field7</th>
<th>field8</th>
<th>field9</th>
<th>field10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When run through the makeheader utility, the previous section of a signal table will appear as follows in
the resulting header file:

```
#define GRIPPER_OPEN 1
```
#define GRIPPER_CLOSE 0
#define GRIPPER_HAND 0

Creating the Signal Header File

As described earlier, the CIMServer automatically assigns each signal a number based on the order in which the signals are defined in the signal table, starting from 0. In a CODE application process, signals are referenced using this numerical index into the signal table.

CODE includes a utility (makeheader) which automatically generates a header file in which the logical name is mapped (using #define’s) into the corresponding numerical index into the signal table. By including this header file in an application process and using the defined constant corresponding to the logical name of the signal, the readability of the source code can be improved.

makeheader Usage

Typing makeheader without any argument will generate the following message:

Utility to make header file for CIMServer signal table.
When -Delphi is specified, a PASCAL unit header file will be created, otherwise, C header file will be created.

Usage: makeheader [-Delphi] <signal_table> [header]

As an example, the following code would be generated in a header file from the signal table shown in Figure 1-1 assuming that no other signals were defined:

```c
/*------------------------ sigdefs.h ----------------------*/
#define SIGNAL_1 0
#define SIGNAL_2 1
#define DI_1 2
#define DI_2 3
#define DO_1 4
#define DO_2 5
#define TEACH_KEY_SWITCH 6
/*-----------------END of the Header File -----------------*/
```

To create a header file for the signal table, do the following:

1. Change to the directory where your signal table is located.

2. Type the following command

   ```bash
   $ROBOTOP/bin/makeheader sigtable_file
   ```

   This command creates a header file with the same name as set in the `sigtable_file` with a .h extension. A header file with a different name can also be created by typing the following command:

   ```bash
   makeheader sigtable_file myheader.h
   ```

   IMPORTANT: If the signal table is not in the current directory, you must include the appropriate (absolute or relative) path name of the signal table.
After a header file has been generated, it should be included (as a header file) in all CODE application processes which interact with the signals.
Chapter 2

Mechanism Motor Homing (MMH)

This chapter will provide information on the tools used for generating MMH files for the currently supported motion cards. A separate graphical user interface tool is required depending on the motion card you intend to utilize. These tools include the MMHUtilDSP, the MMHUtilIPC, and the MMHUtilXMP. The following sections will discuss the usage and menu items for each of these tools in detail.

MMHUtilIPC Generation

To generate an MMH file for the IPC card, do the following:

1. Select the “Mech” button in the MMHUtilIPC, to create a new mechanism instance on the form. The maximum number of mechanisms you can have in a MMH file is 16.
2. To edit the mechanism information, select the mechanism by left-clicking on the mechanism instance. Right-clicking will bring up a popup menu. The two available options are “Delete” and “Change Mech Name”. “Delete” will remove the selected mechanism instance. “Change Mech Name” will bring up a name edit dialog. This allows the user to enter a new mechanism name.
3. Select the “joint” button in the MMHUtilIPC to create a new joint instance on the form. The maximum number of joints that you can have in a MMH file is 16.
4. To edit the joint information, select the joint by left-clicking on the joint instance. Right-clicking will bring up a popup menu. There are three available options “Delete”, “Joint Property” and “Change Joint Name”. “Delete” will remove the selected joint instance. “Change Joint Name” will bring up a name edit dialog. This allows the user to enter a new joint name. Selecting “Joint Property” will bring up a new “joint property” dialog. The “joint property” dialog allows the user to change the conversion ratio and the home sequence.
5. Select the “Motor” button in the MMHUtilIPC to create a new motor instance on the form. The maximum number of motors that you can have in a MMH file is 16.
6. To edit the motor information, select the motor by left-clicking on the motor instance. Right-clicking will bring up a popup menu. The two available options are “Delete” and “Change Motor Type”. “Delete” will remove the selected motor instance. “Change
7. The next step will be to connect some of the instances together. For example: the user might want to connect joint1 and joint2 to mech1. This can be accomplished by left-clicking on the mech1 instance, pressing and holding the shift bar, and left-clicking on the joint1 and joint2 instances to finish connections. The user can do the same to make a connection between joints and motors. When making these types of connections, the higher level instances must be selected first. For example: selecting mech1, pressing and holding the shift bar, and then left-clicking on joint1 would be correct. However, selecting joint1, pressing and holding the shift bar, and then left-clicking on mech1 will not work.

**MMHUtilIPC Menu Items**

Under the “File” menu, there are 5 items: “New”, “Open”, “Save”, “Save As” and “Exit”. Normally, an MMH file will have the “. MMH” extension. This naming structure is however not strictly enforced in MMHUtilIPC, and allows the user to name files as desired.

**Note:**
The MMHUtilIPC gives the user full design freedom in generating MMH files. It is possible for the user to generate an MMH file which has 3 motors mapped to the same joint, or with two mechanisms mapped to the same joint. However, these types of file designs will not be usable for actual control purposes. The init.ipc would not be able to interpret this type of file correctly. For more information on IPC card MMH files, please refer to the IPC device interface release notes.

**MMHUtilDSP Generation**

To generate an MMH file for the DSP card, do the following:

1. To add a card to the MMH file, select “Add a card” under the “File” Menu, or select the “Add” icon. This action will bring up a child window within the main window of the MMHUtilDSP.
2. To edit the card information, select the card by left-clicking on the card instance. The card instance will appear to be pressed down. Right-clicking on the card will bring up a popup menu. “Change Card Name” is the only option available. “Change Card Name” will bring up a name edit dialog. This allows the user to enter a new card name.
3. Select the “Mech” button in the MMHUtilDSP to create a new mechanism instance on the form. The maximum number of mechanisms you can have in a MMH file is 16.
4. To edit the mechanism information, select the mechanism by left-clicking on the mechanism instance. Right-clicking will bring up a popup menu. There are two available options “Delete” and “Change Mech Name”. “Delete” will remove the
selected mechanism instance. “Change Mech Name” will bring up a name edit dialog. This allows the user to enter a new mechanism name.

5. Select the “Joint” button in the MMHUtilDSP to create a new joint instance on the form. The maximum number of joints you can have in a MMH file is 16.

6. To edit the joint information, select the joint by left-clicking on the joint instance. Right-clicking will bring up a popup menu. The three available options are “Delete”, “Joint Property”, and “Change Axis ID”. “Delete” will remove the selected joint instance. “Change Axis ID” will bring up a name edit dialog. This allows the user to enter a new axis ID. Selecting “Joint Property” will bring up a new “Home property and more…” dialog box. The user will be able to enter the new home commands, change the conversion ratio and home sequence, and select the AxisPresent and AxisType settings from this dialog box. Select the “Add” button to add new home command. Select the “Remove” button to remove the existing home command. The program will automatically remove the last command. Select “Edit” to bring up the “Home command property” dialog. This allows the user to setup home type, home latch, home velocity, and etc....

7. Select the “Motor” button in the MMHUtilDSP to create a new motor instance on the form. The maximum number of motors you can have in a MMH file is 16.

8. To edit the motor information, select the motor by left-clicking on the motor instance. Right-clicking will bring up a popup menu. “Delete” is the only option available. “Delete” will remove the selected motor instance.

9. The next step will be to connect some of the instances together. For example: the user might want to connect mech1 and mech2 to card1. This can be accomplished by selecting card1, pressing and holding the shift bar, and then left-clicking on the mech1 and mech2 instances to finish connection. The user can do the same to connect between mechanisms and joints, or joints and motors. When making these types of connections, the higher level instances must be selected first. For example: selecting card1, pressing and holding the shift bar, and then left-clicking on mech1 will be correct. However, selecting mech1, pressing and holding the shift bar, and then left-clicking on card1 will not work.

10. As a MDI application, MMHUtilDSP allows users to work on several card instances the same time. When the user chooses “Save” or “Save As”, every card instance on all the child windows will be saved into the MMH file. So, make sure to close the child windows that you do not want to be saved before executing the save command.

**MMHUtilDSP Menu Items**

Under the “File” menu, there are 6 items: “Add New Card”, “Open”, “Save”, “Save As”, “Clear” and “Exit”. Normally, a MMH file will have the “.MMH” extension. This naming structure is however not enforced in the MMHUtilDSP, and allows the user to name the files as desired. When the user opens a MMH file, the MMHUtilDSP will open child windows based on how many cards are present in the MMH file.
Under the “Window” menu, there are 5 items: “Cascade”, “Tile”, “Arrange Icons” and “Minimize All”. These menu items perform normal windows management operations.

**Note:**
MMHUtilDSP gives the user full design freedom in generating MMH files. It is possible for the user to generate an MMH file which has 3 motors mapped to the same joint, or with two mechanisms mapped to the same joint. However, these types of file designs will not be useable for actual control purposes. The init.ipc would not be able to interpret this type of file correctly. For more information on DSP card MMH files, please refer to the DSP device interface release notes.

**MMHUtilXMP Generation**

To generate an MMH file for the XMP card, do the following:

1. Select the “Card” button in the MMHUtilXMP to create a card instance on the form. Only one card is allowed in the MMH file.
2. To edit the card information, left-click on the card instance. The card instance will appear to be pressed down. Right-clicking on the card will bring up a popup menu. There is one available option “Change Card Name”. “Change Card Name” will bring up a name edit dialog. This allows the user to enter a new card name.
3. Select the “Mech” button in the MMHUtilXMP to create a new mechanism instance on the form. The maximum number of mechanisms you can have in a MMH file is 16.
4. To edit the mechanism information, select the mechanism by left-clicking on the mechanism instance. Right-clicking will bring up a popup menu. “Delete” and “Change Mech Name” are the two available options. “Delete” will remove the selected mechanism instance. “Change Mech Name” will bring up a name edit dialog. This allows the user to enter a new mechanism name.
5. Select the “Joint” button in the MMHUtilXMP to create a new joint instance on the form. The maximum number of joints you can have in a MMH file is 16.
6. To edit the joint information, left-click on the joint instance. Right-clicking will bring up a popup menu. The three available options are “Delete”, “Joint Property”, and “Change Joint Name”. “Delete” will remove the selected joint instance. “Change Joint Name” will bring up a name edit dialog. This allows the user to enter a new joint name. Selecting “Joint Property” will bring up a new “joint property” dialog box. The “joint property” dialog box allows the user to change the conversion ratio and home sequence. Select the “Add” button to add a new home command. Select the “Remove” button to remove an existing home command. The program will automatically remove the last command. Select “Edit” to bring up the “Home command property” dialog. This allows the user to setup home type, home latch, home velocity, and etc....
7. Select the “Motor” button in the MMHUtilXMP to create a new motor instance on the form. The maximum number of motors that you can have in a MMH file is 16.
8. To edit the motor information, left-click on the motor instance. Right-clicking will bring up a popup menu. “Delete” is the only option available. “Delete” will remove the selected motor instance.

The next step will be to connect some of the instances together. For example: the user might want to
connect mech1 and mech2 to card1. This can be accomplished by selecting card1, pressing and holding the shift bar, then left-clicking on the mech1 and mech2 instances to finish the connection. The user can do the same to make a connection between mechanisms and joints, or joints and motors. When making these types of connections, the higher level instances must be selected first. For example: selecting card1, pressing and holding the shift bar, then left-clicking on mech1 would be correct. However, selecting mech1, pressing and holding the shift bar, and then left-clicking on card1 will not work.

**MMHUtilXMP Menu Items**

Under the "File" menu, there are five items, "Open", "Save", "Save As", and "Exit." Normally, an MMH file will have the ".MMH" extension. This naming structure is, however, not strictly enforced in MMHUtilXMP and allows the user to name the files as desired.

**NOTE:** MMHUtilXMP gives the user full design freedom in generating MMH files. It is possible for the user to generate an MMH file which has three motors mapped to the same joint, or with two mechanisms mapped to the same joint. However, these types of file designs will not be useable for actual control purposes. The init.ipc would not be able to interpret this type of file correctly. For more information on XMP care MMH files, please refer to the DSP device interface release notes.

**PMAC to OAC Conversion**

**Generate a PMAC Data File**

- Run the PMAC Executive program from Delta Tau to set up your system
- Select the ‘Backup’ menu
- Select the ‘Save configuration’ option to save configuration to disk
- Enter the name to be used and save the file

**Keep only the I-Variable and Motor Definitions from the PMAC Data File**

- Open the Data file with a text editor such as Notepad
- Keep the commented lines (first few lines starting with a semicolon ‘;’) containing the creation date and the PROM Version but delete everything else between that and the I-Variable definitions (I-Variable definitions starts with I1=___)
- Go to the end of the I-Variable definitions Variables (there are 1024 of each variable) and delete everything, including all of the Q, P, and M-Variables down to the motor definitions (Motor definitions start with #1->___) and are located at the bottom of the file.

**Add Card and Motor Definitions**

- Add ‘%@ 1.40’ as the first line of the data file
- Add a line to define the card and place it after the commented lines which contain the creation date and PROM version. The two possibilities are shown below, c1 indicates card #1, c2 would be used
for a second card, etc.;
- \[c1 = \text{PMAC 8 15 1}' \] for an 8 axis PMAC card or
- \[c1 = \text{PMAC 4 15 1}' \] for a 4 axis PMAC card
- Next add a motor definition section for each motor, including the desired homing options
  - For the motor that is physically wired to the PMAC card as motor #1, start the definition with
    \['mo1 = 50.0 \text{ home1 @ 0.0000 : c1}', \] where 50 is the number used at the bottom of the data file were \ ['#1->50y'
  - The method to be used for homing is described using the home keyword. Below are options to use for homing;
    - Use \('\text{home-1}' \) for a motor that is to be hommed at it’s current position
    - Use \('\text{home0}' \) for a motor that doesn’t need to be hommed
    - Use \('\text{home1}' \) to home an axis in the first homing sequence
    - Use \('\text{home2}' \) to home an axis in a second sequence of homing
    - Use \('\text{home3}' \) to home an axis in a third sequence, etc
  - The \('@0.0000' \) indicates what the position is when homing has been completed. 0.000 causes the home position to be called 0.
  - The \('c1' \) at the end of the definition indicates that this is motor#1 on Card #1.
- The second line to add for the motor definition describes the type of amplifier used;
  - \('\text{amp=0}' \) should be used for a differential amplifier.
  - \('\text{amp=1}' \) should be used for a Non-differential amplifier.
- The third and final line for the motor definition describes the user units used.
  - \('\text{Unit1=0}' \) indicates cnts/user unit
  - \('\text{Unit1=1}' \) indicates cnts/cnt
  - \('\text{Unit1=2}' \) indicates cnts/inch
  - \('\text{Unit1=3}' \) indicates cnts/mm
  - \('\text{Unit1=4}' \) indicates cnts/degree
  - \('\text{Unit1=5}' \) indicates cnts/radian
- Delete the PMAC motor definitions at the bottom of the file (lines starting with \('#1->', \'#2->', etc.)

**Add Mechanism Definitions**

- For each mechanism definition, start with the line \('me1 = RT3000' \) where the mechanism name is the same as in the corresponding workcell file (RT3000 in this example) for mechanism #1. This should be located directly after the card and motor definitions.
- The following lines will set up the motor to joint mappings:
  - Associate the joint # from the workcell file with a physical motor. An example would be \('j1 = T 1.0000*mo2:c1' \) where Joint #0 is named \('T' \) in the workcell file and corresponds to motor #2 on Card #1. The joint number, \(j1 \) in this example, is generated in the Workcell file and has a value of 1 higher than that found in the workcell file.
  - Repeat the previous step for each joint and motor pair in the mechanism.
- Repeat the mechanism definition for each mechanism as set up in the workcell file.
Remove Unused Variables

Remove I-Variables for motors that won't be used. I-Variables are set up so that basically, all variables for axis #1 control will be located between I100 to I185, with axis 2 being between I200 to I285, etc. The I-Variables from I186 to I199 are used for coordinate system #1 (the first mechanism) and I286 to I299 are for coordinate system #2, etc. For example, on an 8 axis card that is only using the first five motors, delete I-Variables from I600 to I899. Leave all variables in the 900 range since they store other information such as homing methods. Delete I-Variables from I286 to I299 for coordinate system #2 or 2nd mechanism, I386 to I399 for coordinate systems #3, etc. if these coordinate systems will not be used.

Using the 'Other' Section

Additional information can be added in a section at the bottom of the OAC file under a section labeled 'Other.'
Chapter 3

CODE Kinematics

Introduction

A mechanism, such as a robot or machine tool, is a collection of links and independent joints, usually connected in a serial fashion and numbered sequentially from the base (primal) joint to the terminal (distal) joint. The independent joints in a mechanism are referred to as the degrees-of-freedom, or simply dof, of the mechanism. With CODE, you can quickly define independent joints.

Many commercial robots have secondary linkages which serve to actuate the independent joints. In CODE terminology, these are called dependent joints because their normal purpose is to actuate an independent joint to a specified joint value.

An example of a set of dependent joints is a ball screw and associated four bar linkage used to rotate a primary robot link about a revolute joint. When you want to simulate more realistic looking robots, you can define secondary dependent joints with the programming interfaces discussed in the following sections.

Reconfiguring a Mechanism: Forward and Inverse Kinematics

Mechanisms can be reconfigured by one of two methods. The first method involves specifying the joint angles for a new robot configuration (forward kinematics). The second method involves specifying a desired Cartesian reference position and orientation (pose) of some tool attached to the end of the robot (as defined by a tool or terminal control frame [TCF]), and then applying a solution set to determine the joint angles necessary to move the TCF to the desired pose (inverse kinematics).

In CODE, inverse kinematics (term shortened to invkin in this chapter) have been provided for several classes of serial mechanisms. Since these invkin solutions are generalized to handle a large class of robots, they may generate solutions which are not valid for a specific mechanism model. In order to improve the invkin computational efficiency, you may want to develop a customized invkin solution for your specific manipulator. This is important when operating in an on-line mode in which trajectory generation requires a significant portion of the available CPU time.

For example, some 6-axis robot invkins will have up to eight sets of joint solutions because the solution process does not impose joint limits until after the invkin generates all possible solutions. Since many 6-
axis commercial robots have only one or two invkin solutions (due to joint limits), you can generate a customized invkin solution for a specific robot model, eliminating all solutions that fall outside the joint limits to improve efficiency.
The CIMServer has several kinematics capabilities, including exact invkin, numerical invkin, and forward kinematics. The next two sections describe these kinematics capabilities.

**Exact Inverse Kinematics**

You can access exact invkin through either CODE libraries and automatic matching procedures, or through a custom invkin solution created by a user who understands robot forward and inverse kinematic theory (an expert user).
The CIMServer’s open architectures for mechanism kinematics (along with the automatic procedures for selecting and matching library inverse kinematics to robots or other mechanisms that you build) makes CODE kinematics easy to use for those not familiar with robot kinematic theory.

**Numerical Inverse Kinematics**

For mechanisms without an exact invkin solution you can activate a numerical invkin procedure. A numerical inverse kinematics solution is available for mechanisms with 2 or more independent, serial joints.
You can enter joint scaling factors to weight the importance of the independent joints relative to each other in a CODE-provided routine. This feature is particularly important for redundant robots having greater than 6 dof. In addition, you can enter a customized numerical invkin routine through the custom robot interface.

**CODE Kinematic Interfaces**

Once you have modeled a mechanism, you can specify the invkin as autodh, custom, or numerical by selecting the appropriate option in the Node Inspector. (For more information on using this panel, see the CIMTools Reference manual.) CODE provides several kinematic interfaces that an expert user can use to program robot kinematics. These interfaces use four routines in the file $ROBOTOP/lib/cimetrix/custom/user_kin.c which is provided in source code form. The routines are described as follows:

branch_DH_invkin(command_entry *q) — This routine branches by iclass number (invkin classification number) to an exact invkin solution for a serial mechanism. The DH invkin routines return the joint values of a robot required to place the frame of an attached robot tool or sensor at some specified pose in space. This frame is often referred to as the tool or Terminal Control Frame (TCF) which is discussed further in the section “Custom Exact Inverse Kinematics” on page 3-29.

branch_custom_invkin(command_entry *q) — This routine branches by iclass number to custom invkin solutions which may or may not be developed from the DH parameters determined for the custom robot.
func_dep_jnt(command_entry *q, jnt_to_check, set_to_limit) — This routine branches by \texttt{fclass} number (forward kinematics classification number) to a user-defined function. The function calculates joint values for joints which are functionally dependent on other joints or which have joint limits which vary by mechanism configuration. The interface to this function is discussed in the section “Functionally Dependent Joints.” Note that an expert user would normally be expected to develop these routines.

redundant(robot *rob, double *scale) — This routine branches by \texttt{rclass} number (redundant/numerical invkin classification number) to a user-defined routine that specifies Jacobian scaling factors for a numerical invkin. This routine is particularly useful for redundant mechanism with kinematically redundant joints. These scaling factors can be applied to a robot with any number of joints. The particular interface to the CODE-specific numerical invkin routine, \texttt{numerical_invkin()}, will be discussed in “Redundant Joints.”

A Note on the Interface Routines

Please note that since you only need to insert your own routines into include files which are then inserted into the main body of these interface routines at compile time, you do \textit{not} need to modify these interface routines. With these routines, you can specify a unique classification number that is used by a case switch in user-modifiable include files to branch to the routine of your choice. The classification numbers are displayed in the Robot Panel dialog box during a CIMTools session and, in most cases, must be uniquely defined for each robot.

\textbf{WARNING!}: Since \texttt{user\_kin.c} is not intended to be modified, back up the source file \texttt{user\_kin.c} (we suggest saving \texttt{user\_kin.c} as \texttt{user\_kin.c.old}) before attempting any modifications to any of the routines in \texttt{user\_kin.c}.

Inverse Kinematics

CODE software provides a library of mechanisms for which exact invkin solutions are available. The library is extensive, and includes many popular models commercially available. If you build a mechanism that is similar to one in the provided library, the CIMServer will automatically calculate the parametric description of the mechanism, and determine whether or not an invkin solution matches one of the library mechanisms. The CIMServer assigns the library \texttt{iclass} number to your mechanism along with other parameters (not documented) which account for differences in link geometry and \textit{zero state} (the robot configuration in which the independent joint values are zero). This is accomplished when you select the \textbf{Auto DH} option from the \textbf{Inverse kinematics} option menu in the CIMTools Robot Panel, the CIMServer automatically generates the DH parameter set for your
mechanism, attempts to match an invkin solution from the CODE libraries, and displays an \textit{iclass}
number which is non-zero if a library match occurs.

If the match fails, the CIMServer will default to using a numerical solution. The Robot Panel will indicate this. The Robot Panel is a unique panel in the Node Inspector that appears when a robot node is selected and entered into the node selector. For more information about the Node Inspector and the Robot Panel, see the \textit{CIMTools Reference} manual.

If your mechanism is not similar enough for the CIMServer to fit a solution, you may be able to generate and use a custom solution. In CODE terminology, a \textit{custom} inverse kinematics solution is a solution that has been customized to suit a specific mechanism, or does not use the same parametric (DH parameter) joint description as those shipped with the CODE invkin solutions library from Cimetrix, Inc.

The expert user has two options for implementing a custom invkin solution for a particular mechanism. The first and most recommended option is discussed in the following section, “CODE DH Inverse Kinematics.” The second option is covered in “Custom Exact Inverse Kinematics” on page 3-29. Cimetrix, Inc. can also develop a solution for you.

\textbf{CODE DH Inverse Kinematics}

In the first and most highly recommended option, you apply a set of modified Denavit-Hartenberg (DH) parameters to generate a general solution that can be entered into the CODE libraries. You begin by having the CIMServer calculate the DH parameters for the robot, then store those parameters in a text file. The API \texttt{CxFileDHParm()} can be used in a CODE application process to call the DH routines to generate the DH parameters, then store them in a user-designated file. The Robot Panel in CIMTools also provides a push button to create DH parameters.

These parameters are used to create a general invkin solution for this new robot class by applying procedures as discussed in numerous robotic texts (as an example, John J. Craig, \textit{Introduction to Robotics: Mechanics and Control}, 2nd edition, Addison-Wesley, 1989). Sample invkin routines are available in source code form as examples of routine organization and argument usage.

This section is divided into three parts as follows:

- “DH Parameters” describes the Denavit-Hartenberg robot kinematic parameters.

- “Mathematical Procedures for Inverse Kinematics” gives you a general idea of how to build a DH invkin solution.

- “Generating and Integrating Your Solution” gives you several sets of “cookbook-style” steps for creating your own solution and integrating it with the CIMServer.
### DH Parameters

The four DH parameters shown in Figure 2-1 are the minimum parameters required to relate joint frames to each other.

The parameters are defined as follows:

- $a_i$ = Minimum distance between joint $i$ axis ($z_i$) and joint $i-1$ axis ($z_{i-1}$).
- $d_i$ = Distance from minimum distance line ($x_{i-1}$ axis) to origin of $i$th joint frame measured along $z_i$ axis.
- $\alpha_i$ = Angle between $z_i$ and $z_{i-1}$ measured about previous joint frame $x_{i-1}$ axis.
- $\theta_i$ = Angle about $z_i$ joint axis which rotates $x_{i-1}$ to $x_i$ axis in right hand sense.

The $x_i$ axis is the minimum distance line defined from $z_i$ to $z_{i+1}$; $z_i$ is defined as the joint rotation or translation axis and the $y_i$ direction is defined by the right hand rule and cross product $y_i = z_i \times x_i$ where bold indicates a unit vector in the axis direction. The origin of each joint frame is defined by the minimum distance line where it intersects the joint axis. For more information about DH parameters and invkin see John J. Craig, *Introduction to Robotics: Mechanics and Control*, 2nd edition, Addison-Wesley, 1989.

### CODE DH Parameter File Format

The DH parameters are stored in user files and/or library files numbered as in Table 2-1.

#### Table 2-1: Allowable iclass ranges for ndof

<table>
<thead>
<tr>
<th>ndof</th>
<th>iclass # reserved by CIMServer</th>
<th>iclass # available to user</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2000 - 2799</td>
<td>2800 - 2999</td>
</tr>
<tr>
<td>3</td>
<td>3000 - 3799</td>
<td>3800 - 3999</td>
</tr>
<tr>
<td>4</td>
<td>4000 - 4799</td>
<td>4800 - 4999</td>
</tr>
<tr>
<td>5</td>
<td>5000 - 5799</td>
<td>5800 - 5999</td>
</tr>
</tbody>
</table>
Because the number of possible invkin solutions will vary depending on robot structure, the maximum number of solutions \( \text{nsoln} \) is specified at the top of the file. The file also contains \( \text{ndof} \) rows (equal to the number of degrees-of-freedom) and eight columns in the order: \( \text{jtype, a, aflg, d, dflg, } \theta, \alpha, \text{ and alphaflg} \).

Joint type \( \text{jtype} \) specifies whether a joint is rotational (revolute) or translational (prismatic); \( \text{jtype} = 1 \) represents a revolute joint while \( \text{jtype} = 2 \) represents a translational joint. DH parameters \( (a, d, \theta, \alpha) \) are automatically created for each of the joints.

The DH parameter flags \( (\text{aflg, dflg, and alphaflg}) \) determine the generality of the custom invkin and are not to be used in the invkin routine itself. These flags will normally be set to default values of one. These flags are automatically used by the CIMServer’s matching routines to determine if a user’s simulation robot has a DH set that matches a library invkin robot.

For example, if a newly developed invkin routine is not bound by particular values of \( a_2 \) (it is not unusual for a robot invkin solution to require that \( a_2 \) only have values other than zero), and does not fail if \( a_2 \) is zero or non-zero, then the DH parameter flag \( \text{aflg} \) for joint 2 should be set to zero. In this way robots having any DH parameter \( a_2 \) can be matched to this particular invkin.

Code Listing 2-1 shows the required DH file format in parameter form; of course, the actual constant values would replace the parameters shown for any given robot (see Code Listing 2-3 for an example).

Since the CIMServer's auto invkin procedures assume that the robot base frame is the same as the robot's first independent joint frame, the first row of DH parameters contains only zero.

The CIMServer internally handles user-specified robot base frames that do not coincide with the first joint frame. This feature gives you full flexibility to locate robot base frames in any desired location and allows you to match the simulation robot to a physical robot being controlled by the CIMServer.

```
maximum number of solutions

nsoln

<table>
<thead>
<tr>
<th>jtype</th>
<th>a</th>
<th>aflg</th>
<th>d</th>
<th>dflg</th>
<th>theta</th>
<th>alpha</th>
<th>alphaflg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000</td>
<td>1</td>
<td>0.00000</td>
<td>1</td>
<td>0.0000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
<td>d2</td>
<td>1</td>
<td>theta2alpha2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a3</td>
<td>1</td>
<td>d3</td>
<td>1</td>
<td>theta3alpha3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a4</td>
<td>1</td>
<td>d4</td>
<td>1</td>
<td>theta4alpha4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

# Note that when user robot DH parm values compared to invkin
# file DH parm values above, it must be true that:
#
# DH parm theta & alpha must be in RADIANS
#
# jtype: 1 = ROTATE 2 = TRANSLATE "joint type"
#
# aflg: 0 = ANY VALUE for DH parm a is OK
# 1 = user robot DH parm a must be ZERO if file a is ZERO. If file DH a NONZERO, then user DH a must be NONZERO
#
# dflg: 0 = ANY VALUE for DH parm d is OK
# 1 = user robot DH parm d must be ZERO if file d is ZERO. If file DH d NONZERO, then user DH d must be NONZERO
#
# alphaflg: 0 = ANY VALUE for DH parm alpha is OK
# 1 = EXACT MATCH or PI difference allowed for DH parm alpha
# 2 = ZERO or PI NOT ALLOWED for DH parm alpha
```
**WARNING!**: When you use CODE to control a physical robot in an actual cell, the robot model must have the same base frame as the physical robot. The inverse kinematics procedures in commercial robots use a terminal interface frame (TIF), usually the last joint frame of the physical robot, and relate this frame to a unique base frame to determine the joint values for desired robot configurations.

The only requirement for joints is that robot models must have their joint frames oriented such that the joint axis for rotational (revolute) joints be collinear with the frame z axis. In the case of translational (prismatic) joints, it is only necessary that the z axis be parallel to the physical joint, although you should choose a joint z axis that is within proximity of the physical components that move the translational link.

For revolute joints, the DH parameter $\theta$ is variable, while for translational joints, $d$ is variable. The following homogeneous transformation (Figure 2-2), which describes the pose of joint frame $i$ relative to joint frame $i-1$, is shown in column form to agree with most literature expositions (the C language actually stores matrix vectors row by row in contrast to FORTRAN which stores them by column). The elements in the first three columns are direction cosines for the $x_i$, $y_i$, and $z_i$ frame unit vectors resolved into the $x_{i-1}$, $y_{i-1}$, and $z_{i-1}$ axes of the joint $i-1$ frame. The fourth column is the origin of joint frame $i$ resolved into joint frame $i-1$.

If joint $i$ is revolute, $d_i$ remains constant while $\theta_i$ will vary; if translational, $\theta_i$ remains constant while $d_i$ will vary. The joint value shown in the homogeneous transformation must be modified to account for joint changes from the robot configuration in which the DH parameters were originally derived.

The DH parameter values typically represent values of the robot in its zero state and do not reflect the joint values when the robot is moved to some non-zero configuration. In the non-zero configuration the joint values must be modified as the joints vary.

For example, if joint $i$ is revolute and, at a new configuration, the encoder indicates a joint value of 60, and if the DH value is 45 when the joint encoder indicated a zero joint value, then the joint value entered would be $\theta_i = 60 + 45 = 105$. $d_i$ would be modified similarly for a translational joint.

**Mathematical Procedures for Inverse Kinematics**

The procedure for determining an invkin solution for a particular robot is defined by the transformation operations:

$$T = B A_1 A_2 ... A_{n-1} A_n G$$

where
\( A_i = \) Joint i transformation  
\( G = \) Gripper or tool frame relative to last joint frame of robot  
\( B = \) Base frame of robot described relative to world frame  
\( T = \) Target frame for gripper described in world frame

The invkin problem becomes one of solving the following equation for the unknown joint values
\[
D = B^{-1} T G^{-1} = A_1 A_2 ... A_{n-1} A_n
\]

where \( D \) defines the desired pose of the distal joint frame relative to the robot base frame. In the autodh procedures the base frame of the invkin robot is the same as the first joint frame and the CIMServer provides you with the desired pose of the distal joint frame relative to the first joint frame. The user-developed invkin routine must return all possible sets of joint solutions and the number of these solutions. The CIMServer will then check all solutions against the joint limits, calling the forward kinematics routines to determine joint dependencies and to determine variable joint limits from other user routines as discussed in the section “Forward Kinematics and Functional Dependencies.”

**Saving DH Parameters with File DH**

You are now ready to build and integrate your own custom solution. The first thing you need to do, as indicated earlier, is determine and save the DH parameters for your robot.

1. Model the kinematics of your mechanism using CIMTools. For more information about the CIMTools, see CIMTools Reference manual.

   We strongly recommend that the robot’s zero state be defined such that the links are configured at 0° or 90° to each other when possible. This simplifies the form of the homogeneous transformations used in solving the inverse kinematics.

   The iclass number and joint letter sequence (R for revolute joints and P for prismatic joints [see example Code Listing 2-3: DH File for RPPR_.4001]) are used together to uniquely identify each DH solution available to the CIMServer. DH parameter files are named letters.iclass. CODE searches the file DHPARAMETERS to find the joint sequence (from the letter sequence) and file name of each available solution.

   An example of the contents of the DHPARAMETERS file is shown below.

   ```
   PP____.2001  
   PPP____.3001  
   RRR____.3002  
   PPR____.3003  
   RPPR____.4001  
   RRRP____.4002  
   RPRR____.4003  
   PPFR____.4004  
   PRRR____.4005  
   PPFR____.4006  
   RPPP____.4007  
   PPPP____.4008  
   RRPR____.4009  
   PRPR____.4010  
   PRPR____.4011  
   RPPP____.4012  
   RPPR____.4013
   ```
2. Choose an iclass number for your solution.
   a. Use the ranges listed in Table 2-1 (page 3-12) to pick an appropriate number.
   b. Search the \$ROBTOP/lib/cimetrix/DH/DPARAMETERS file to ensure that your iclass number is unique.

   **NOTE:** ROBTOP is the directory under which the CODE software was installed.

3. Determine the letter sequence (P for prismatic, R for revolute) of the robot's independent joints (see Code Listing 2-2 for examples).

4. Enter the DH file name and iclass number into the file DPARAMETERS in the directory \$ROBTOP/lib/cimetrix/DH.

5. Create the DH parameter file by following these steps:
   a. While the CIMServer and CIMTools are running, open the Node Inspector and update it to work on the robot you want to create a solution for. It becomes the Robot Panel.
   b. Click on the **Save DH Parameters** button in the Robot Panel.

      The **Save DH Parameters** file name dialog box will appear on your screen.
   c. Change directory to \$ROBTOP/lib/cimetrix/DH in the **Directories** scroll list.
   d. Type where name means a file name corresponding to the number and types of joints in the robot in the **Selection** field.
e. Click on **Save** button.

For example, the file name for a four axis Seiko robot having a first joint which is revolute (R), second and third joints both translational (or prismatic, P), and the fourth revolute (R), and for which you have specified an iclass number of 4001 would be RPPR__.4001. The DH file for the robot would appear as follows:

```
maximum number of solutions
0
jtype a aflg d dflg theta alpha alphaflg
1 0.0000 1 0.0000 1 0.0000 0.0000 1
2 0.0000 1 0.0000 1 1.5707 0.0000 1
2 0.0000 1 0.0000 1 0.0000 1.5707 1
1 0.0000 1 0.0000 1 0.0000 -1.5707 1
```

#--------------------------------------------------------------
#Note that when user robot DH parm values compared to invkin file DH parm
#values above, it must be true that:
#    # DH parm theta & alpha must be in RADIANS
#    # jtype: 1 = ROTATE  2 = TRANSLATE "joint type"
#    # aflg: 0 = ANY VALUE for DH parm a is OK
#    # 1 = user robot DH parm a must be ZERO if file DH a is
#    #      ZERO. If file DH a NONZERO, then user DH a must be
#    #      NONZERO
#    # dflg: 0 = ANY VALUE for DH parm d is OK
#    # 1 = user robot DH parm d must be ZERO if file DH d is
#    #      ZERO. If file DH d NONZERO, then user DH d must be
#    #      NONZERO
#    # alphaflg: 0 = ANY VALUE for DH parm alpha is OK
#    # 1 = EXACT MATCH or PI difference allowed for DH parm
#        alpha
#    # 2 = ZERO or PI NOT ALLOWED for DH parm alpha
#--------------------------------------------------------------

Code Listing 2-3: DH File for RPPR__.4001

You must **not** modify the DH parameters in the automatically created DH file, but you can modify the maximum number of solutions (shown as 0 on the second line in Code Listing 2-3) and the flag values in the aflg, dflg, and alphaflg columns to either 0 or 1. The API CxFileDHParm can also be used in a CODE application process to call the DH routines to generate the DH parameters, then store them in a user-designated file.

---

## Generating and Integrating Your Solution

This section assumes you know how to generate the inverse kinematics for your own robot, given the DH parameters and the DH file format. If you do not have enough knowledge to do so, you will have to do some outside research and reading, or contact Cimetrix, Inc. for a solution.

This section first lists all of the necessary steps to integrating your solution. In the section that follows, examples and hints will be given in the same order as the steps.

1. Generate the invkin solution C program, for example `myrobot.c`
2. Enter the call to the routine into the include file `user_kin.IK` as a C language case switch statement.

   See Code Listing 2-4 for an example. In the example the invkin solution C routine is denoted by the name `rt_3000_inv`. The include file `user_kin.IK` is found in `$ROBTOP/lib/cimetrix/custom` (ROBTOP is the CODE installation directory) and is user-modifiable.

3. Edit the makefile in `$ROBTOP/lib/cimetrix/custom`, adding the new `myrobot.c` file. The new `myrobot.c` file will compile and link the file to the necessary CODE routines for testing and debugging.

   Code Listing 2-6 shows the appropriate makefile segment.

4. Type `gmake` in the directory `$ROBTOP/lib/cimetrix/custom` to make (compile and link) the new routine.

   Once the debugging has been completed, you now have another useful library robot with an invkin solution that can be used for the inverse kinematics of similar robots.

   NOTE: The best way to debug the invkin routine is to write a forward kinematics routine that computes the `tdis` frame at a particular set of joint angles and then require the invkin to return the same set of joint angles.

```c
/*----------------------user_kin.IK---------------------------------
 Case switch for inverse kinematics solutions. If you use CIMServer
 generated DH parameters to derive your invkin solutions, this is the file
 where you need to add your custom case.

 Here is the guideline for selecting a case number for your invkin
 solution.
 [1] The first number represents the degree of freedom of the robot
 [2] The second number for your robot must be either 8 or 9. Do not use
     other numbers. They may conflict with future software releases. (i.e. 8 or 9
     are reserved for your robot.)
 [3] The last two numbers can be any number between 0-9.

     example:
       3801 is a valid choice for a 3-axis robot
       4951 is a valid choice for a 4-axis robot
       6901 is a valid choice for a 6-axis robot
       3001 is not a good number, it's been used
       3005 is OK, but not recommended, use 3805 or 3905 instead

------------------------------------------------------------------*/

case 2001:
    xy_robot(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok);
    break;

case 3001:
    seikoXm(rob->tdis,&rob->nsoln,rob->soln,rob->soln_ok);
    break;

case 3002:
    tescon(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
             soln_selected, q->inner_tcf,rob->dofold,rob->tcf_to_joint);
    break;
```
case 3003:
    strippit(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4001:
    rt_3000_inv(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
                q->inner_tcf, rob->dofold, rob->tcf_to_joint);
    break;

case 4002:
    scara(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
          soln_selected, q->inner_tcf, rob->dofold, rob->tcf_to_joint);
    break;

case 4003:
    a_510(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
          soln_selected);
    break;

case 4004:
    four_axis_NC(rob->tdis, &rob->nsoln, rob->soln, rob->soln_ok,
                 q->inner_tcf, rob->dofold, rob->tcf_to_joint);
    break;

case 4005:
    scara2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
           soln_selected);
    break;

case 4006:
    seikoXm4(rob->tdis, &rob->nsoln, rob->soln, rob->soln_ok, q->inner_tcf,
              rob->dofold, rob->tcf_to_joint);
    break;

case 4007:
    anorad_NC(rob->tdis, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4008:
    anorad2_NC(rob, rob->tdis, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4009:
    scara3(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
           soln_selected);
    break;

case 4010:
    prpr(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
         rob->dofold, soln_selected);
    break;

case 4011: /* similar to custom kin case 4001 */
    prpr2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
          rob->rob_to_inv, rob->last_robjnt_to_invjnt, 0);
    break;

case 4012:
    cnc1(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
         rob->dofold, soln_selected, q->inner_target);
    break;

case 4013:
    m100r500e1(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;
case 4014:
    a200e2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4015:
    m100r500e2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4016:
    h100si_xxxx(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 4019:
    dekprint(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 5001:
    modulaser(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        rob->dofold, soln_selected);
    break;

case 5002:
    gmfe200(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        rob->dofold, soln_selected);
    break;

case 5003:
    t3735(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5004:
    pana_inv5(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5005:
    generic1(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5006:
    m100r500f1(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5007:
    a200f2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5008:
    generic2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 5009:
    m100r500f2(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok);
    break;

case 5010:
    l100(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
    break;

case 5011:
    m400_5(rob->tdis, rob->parm, &rob->nsoln, rob->soln, rob->soln_ok,
        soln_selected);
break;

case 6001:
    gmfs100_inv(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6002:
    g200(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6003:
    t3786(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6004:
    puma_inv(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6005:
    sa130(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6006:
    SeikoVT_inv(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6007:
    titan_inv(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6008:
    cybo(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

case 6009:
    eh120(rob->tdis,rob->parm,&rob->nsoln,rob->soln,rob->soln_ok,
    rob->dofold,soln_selected);
    break;

Code Listing 2-4: Sample Include File user_kin.IK

Examples and Hints for Integrating Your Solution

The arguments are from the CIMServer’s robot data structure and represent the following:

- rob->tdis: Input terminal displacement matrix corresponding to the D transformation identified earlier.
- tdis is declared as a matrix tdis in your invkin routine.
robor->parm: Input DH parameter values for the robot in the order shown previously and numbered from 0 — the first joint is designated zero to be consistent with C convention — to ndof-1 for the last joint. parm should be declared as double **parm in the invkin routine.

robor->nsoln: The number of invkin solutions returned for this robot and declared as long *nsoln in the invkin routine.

robor->soln: The solutions determined in your invkin routine and returned for each joint in the array rob->soln[rob->nsoln][ndof] where soln is declared as double **soln in the invkin routine.

robor->soln_ok: Vector array of values set to either CX_TRUE or CX_FALSE for each solution returned in the rob->soln array; dimensioned as rob->soln_ok[rob->nsoln] where rob->soln_ok is declared as long *soln_ok in the invkin routine. This array allows you to return only certain solutions from the invkin routine depending on the tdis frame entered.

**WARNING!:** The tdis matrix is provided in C consistent row form. You must recognize this form difference if the invkin solution has assumed a tdis in column form. For example, if the invkin equations require the column element tdis(2,1), then the code must substitute tdis[1][2] into the particular operations required in the invkin equations.

Although these are typically the arguments in a user developed invkin routine, you could pass the entire robot data structure through to get access to other useful parameters such as the robot joint values in the last configuration stored in the robot data structure dofold and accessed for joint i as dofold[i] or rob->dofold[i], depending on how it is passed into the routine. In this way the routines could be customized to handle robot singularities and other special cases. In your routine the robot pointer would be declared robot *rob.

An example of a simple inverse kinematics routine is shown in Code Listing 2-5 for the Seiko RT3000 robot. The "include" files shown at the top of the routine should normally be included.

```c
/********************************************
FUNCTION rt_3000_inv
REMARKS
calculate the 4 joint angles for a scara robot
********************************************/
#include <code/matrix.h>
#include <code/const.h>

void rt_3000_inv(CxMatrix tdis, double **parm, long *nsoln, double **soln,
        long *soln_ok, long inner_tcf, double *dofold, CxMatrix tcf_to_joint)
{...
```
double d1,d2;
double r_target;
double x,y,X,Y;
double gamma,theta_target;

d1=parm[1][1];
d2=parm[2][1];

if(inner_tcf == 3 && (fabs(tdis[2][0]) > CX_EPSILON ||
  fabs(tdis[2][1]) > CX_EPSILON ||
  fabs(tdis[2][2] - 1.0) > CX_EPSILONS)) {
  *nsoln=0;
  return;
}

if(inner_tcf == 3) {
  if(fabs(tdis[3][1]) < CX_EPSILONS && fabs(tdis[3][0]) < CX_EPSILONS)
    soln[0][0]=0.0;
  else
    soln[0][0]=atan2(tdis[3][1],tdis[3][0]);
  soln[0][1]=tdis[3][2] - d1;
  soln[0][2]=sqrtabs(tdis[3][0]*tdis[3][0] + tdis[3][1]*tdis[3][1]) -
    d2;
  soln[0][3]= atan2(-tdis[1][1],-tdis[1][0]) - soln[0][0];
}
else if(inner_tcf == 2) {
  x = tcf_to_joint[3][2];
  y = tcf_to_joint[3][0];
  X = tdis[3][0];
  Y = tdis[3][1];
  r_target = sqrt(X*X + Y*Y);
  gamma = asin(y/r_target);
  theta_target = atan2(tdis[3][1],tdis[3][0]);
  soln[0][1] = tdis[3][2] - tcf_to_joint[3][1] - d1;
  if(fabs(gamma) < CX_EPSILONSS)
    { 
      soln[0][0] = theta_target;
      soln[0][2] = r_target - x - d2;
    }
  else
    { 
      soln[0][0] = theta_target - gamma;
      soln[0][2] = r_target*cos(gamma) - x - d2;
    }
  soln[0][3] = dofold[3];/* set jnt 4 to dofold value */
}
else {
  soln_ok[0] = CX_FALSE;
  *nsoln=0;
  return;
}

/********** set number of solutions and soln flag ***********/
The user_kin.IK file is included in the routine branch_DH_invkin() in the file user_kin.c. Since the file user_kin.c is included in source form, the user can review it to see how the include files are included at compile and link time. To compile and link a user routine, you only need to add the user routine to the makefile to the last line of the CFILES = routines as shown in Code Listing 2-6. If insufficient space exists on the line to add your filename, then add a backslash " \ " to the present line and append a new line with your file. Type “gmake” and the routine will be compiled and linked to the CIMServer.

The segment of the makefile to be edited for new invkin files is listed as follows:

```plaintext
CFILES= PhySignals.c robmain.c threadmain.c user_kin.c \ 
user_motion.c user_cntrl.c
```

TCF’s Attached to Inner Joints

Normally, robot tools will be attached as a child of the last joint, sometimes referred to as the distal joint. The Tool’s Control Frame (TCF) is then used in invkin routines to determine the joint values for a target type move. Given a desired target frame for the TCF, the homogeneous transformation relating the TCF to the last joint frame is used to generate a new target frame for the last joint frame. The invkin routine then determines the joint values that will place the last joint frame at the new target frame. Sensors are often attached to a mechanism for either measuring the rigid-body pose of parts, quality control measurement, or for calibrating mechanism inaccuracy relative to part features. There are cases when it is advantageous to attach these sensors to joints other than the terminal joint. For example, robots with rotational ending joints have difficulty managing the cables which feed outside the robot to cameras mounted on the terminal joints. In many such cases, the robot positioning error introduced by the last joint may be insignificant when compared to that introduced by the inner joints. It may then make sense to mount cameras (or other sensors) to the interior links of the robot or mechanism for calibration purposes. The CIMServer provides the user an interface for accommodating TCF’s mounted to inner joints.

Attaching TCF’s to inner joints means that the invkin procedures must be revised slightly, effectively modifying the routines so that the joint values are computed for a reduced set of joints. The
inner_tcf parameter, a member of the command_entry data structure, can be used by the user to determine if the TCF is attached to the last joint or to an inner joint and is passed as q->inner_tcf through the call to the invkin routine. Its value will be the number of the joint to which the TCF is attached. In most cases its value will be equal to rob->ndof -1.

In addition, the robot data structure parameter tcf_to_joint must also be passed in as rob->tcf_to_joint. This locates the TCF relative to the joint frame identified by the value of inner_tcf.

Since it may not always be possible to close the kinematic loop equation shown in the section "Mathematical Procedures for Inverse Kinematics" for a reduced number of joints, the tdis matrix will not be interpreted as the pose of the last joint frame necessary to move the tool TCF to the target frame. Rather, the actual TCF target is passed to the invkin routine and should be interpreted as tdis relative to the first joint frame of the inverse kinematics robot. This is a different meaning for tdis, which, for the normal robot, represents the desired target of the last joint frame of the robot. It is now the responsibility of the user to use the additional matrix that relates the TCF to the inner joint (tcf_to_joint) and the desired TCF target (tdis) to extract the inner joint values.

Inside the invkin routine the user should compare the value of inner_tcf to ndof-1 as outlined in the partial code listing which follows. This code demonstrates how the routine can be organized for conventional TCF's and inner TCF's.

```c
long my_routine(char* normal arguments, long ndof, double *dofold,
               long inner_tcf, matrix tcf_to_joint)
{
    long i;
    "other normal declarations"

    if(inner_tcf == ndof -1) /* TCF attached to distal joint */
    {
        "perform invkin calculations for all ndof-1 joints"
    }
    else /* TCF attached to inner joint */
    {
        "perform invkin calculations for joints 0 to and including the inner_tcf joint"

        /* now set remaining joint values to dofold values */
        for(i = inner_tcf + 1; i < ndof; i++)
        {
            soln["soln number"](i) = dofold[i];
        }
    }

    "terminate routine as normal"
}
```

The user will note that all joint values beyond the "inner joint" were set to their old values saved in the robot data structure as dofold.

The API CxGetInnerTcfOffset() can be used to get the achieved target position and orientation offsets from the desired target. In many cases the achieved target will reach the origin of the desired target, but not the orientation. Refer to the Programmer’s Reference Manual for more details on how to use this API. Note that the user should set a blend policy of CX_MOVE_WAIT to get the target
offsets using this API, since the target offsets only have meaning when determined at end of motion of
the move.

Suggested inner TCF implementation for automatic inverse kinematics:

- Pass in additional parameters: inner_tcf, ndof, dofold, and tcf_to_joint.
- Branch invkin solutions by inner_tcf value.
- Solve for reduced set of joint values.
- Set joint values of joints beyond inner_tcf to dofold values.
- In the CODE Application process, set the blend policy to CX_MOVE_WAIT.
- To get target offsets the call to CxGetInnerTcfOffset() should be placed immediately
  after the move command in the CODE Application process.

**Targets Attached to Inner Joints**

Normally, a target will not be attached to a joint (a child of) on the robot. Normal exceptions are when
the target is specified as a move relative to the tool TCF, or the target (part) is attached to the last table
joint of a machine tool as in a three axis mill. But there are some cases where it may be useful to attach
a target to an inner joint of a robot. An example would be the case where an A, B, or C auxiliary axis is
added to the table of a machine tool, yet it is necessary to make moves relative to nodes attached to the
X or Y table axis, neither of which will now terminate the table joints of the machine tool.

Procedures have been added to CODE kinematics that will enable the user to customize the kinematics
in such circumstances. CODE will automatically determine whether the target is attached to an inner
joint of the robot. The user can ascertain this by examining the q->inner_target parameter.

If the inner_target value is -1, then the target node is not a child of an inner robot joint for serial
robots. For NC robots this should be interpreted as the part attached normally to the last table joint.
Values of 0 or greater indicate the joint which is parent of the target node. The user will then have to use
a subset of the joints to solve the inverse kinematics problem. The user should use joints after the
inner_target joint to solve the invkin problem. For example, if i is the inner_target joint, then the
following subset of joint transformations would be used to solve the invkin problem:

\[
A_{i+1}A_{i+2}...A_{n-1}A_n
\]

If the user is using DH parameters generated by the CODE server, then one modification must be made
to the DH values for the inner target joint. Since a subset of frames are being used, the inner target joint
must be treated as if it were the first joint in the sequence. This requires that the user generate the Ai+
frame above using all zero state values for the DH parameters. Remember that normally the DH values
for the first joint in the serial joint sequence are all zero; thus, the user must also set the DH values for
the inner target joint to zero also. The remaining joints use the same DH parameter values without
modification.

In the inverse kinematics routine the user should assign the joints interior to and including the inner joint
their rob->do-fold values. If these joints are being treated as auxiliary values, then CODE will assign
their values outside the inverse kinematics routine correctly.

CODE treats a NC robot as a serial robot in the relative sense by counting backwards from the last joint
of the base branch to the terminal joint of the tool (upper) branch. Thus, in the relative sense, q->
inner_target has the same meaning as for a normal serial robot.
CODE will pass the tdis matrix to the inverse kinematics routine relative to the inner target joint, rather than the first or zero joint of the robot.

Attaching targets to inner joints means that the invkin procedures must be revised slightly, effectively modifying the routines so that the joint values are computed for a reduced set of joints. The inner_target parameter, a member of the command_entry data structure, can be used by the user to determine if the target is attached to an inner joint and then pass q->inner_target through the call to the invkin routine. Its value will be the number of the joint to which the target is attached. In most cases its value will be equal to -1, which indicates that the target is not attached to a serial joint or, for NC robots, that it is attached to the last table joint. This code demonstrates how the routine can be organized for conventional targets and inner targets.

```c
long my_routine("normal arguments",long ndof,double *dofold,
               long inner_target)
{
    long i;
    "other normal declarations"

    if(inner_target == -1)/* normal target */
    {
        "perform invkin calculations for all ndof-1 joints"
    }
    else /* target attached to inner joint */
    {
        "perform invkin calculations for joints following inner target
         to last joint to which tool is attached"

        /* now set interior joint values to dofold values */
        for(i = 0; i <= inner_target; i++)
        {
            soln["soln number"][i] = dofold[i];
        }
    }

    "terminate routine as normal"
}
```

The user will note that all joint values interior to and including the "inner target joint" were set to their old values saved in the robot data structure as dofold. For the special case where an inner TCF is also used, the user should refer to the preceding section for the implementation.

Suggested inner target implementation for automatic inverse kinematics:

- Pass in additional parameters: inner_target, ndof, and dofold.
- Branch invkin solutions by inner_target value.
- Solve for reduced set of joint values.
- Set joint values of joints beyond interior to and including inner_target to dofold values.
Debugging Your Inverse Kinematics Solution

Programming errors will compound the difficulty of adding a new DH invkin to the library. Some of the difficulty will be caused by errors in your routine that make the CIMServer’s routines fail elsewhere in the code, such as the overwriting of array space. Be very careful not to make these common mistakes.

Since the makefile allows you to turn on the symbolic debugger option flag (e.g. Unix command: `setenv GCFLAGS -g`), you will be able to break into your routine and step through line-by-line during program execution. This should assist your debugging.

Since the DH parameter values passed through the parm array are the basis for your invkin solution, you need to understand what values are actually passed to your routine. If you build a new user robot for which no DH invkin solution exists, then you may choose to build a new DH invkin. The DH parameters file that you generate for this robot is the file that will be passed to your new invkin solution. Your invkin solution should be based on this set of DH parameters.

An exception occurs when another robot is built that is slightly different in joint polarity, link lengths, or zero state, but the robot still falls into one of the robot classes contained in the library. In this case the DH parameters passed into the library invkin are the original DH parameters of the library robot, modified to account for different link lengths of the new user robot (a and d values are modified to those of the new user robot). These procedures are transparent to the user since he or she will not normally modify the library invkin solutions.

The CIMServer must account, in some automatic way, for user developed invkin solutions that may return joint values which are not constrained by joint limits and which, in the case of revolute joints, may not apply standard rules for assigning joint values by quadrant. For example, 220° and -140° may refer to the same quadrant location, but have a quite different physical implication when used to move a physical joint.

Other differences occur when the CIMServer automatically assigns the user robot the invkin of a DH type library robot. In such cases the different joint polarities and differences in the robot zero state may cause the joint values calculated in the library invkin to be quite different from those displayed in the joint panels.

To harmonize the joint value calculations, solutions in the CIMServer are first modified for zero state differences and joint polarity. These differences will not confuse the picture for custom robots since the zero state and joint polarity differences will only apply for DH invkin solutions.

All revolute joint solutions are next adjusted such that their values fall in the range $-180 \leq \theta \leq 180$. The new revolute joint values are then compared to the previous joint values (temporarily placed into this same range) and the minimum and maximum joint changes calculated in an absolute sense. For example, consider a previous joint value of 130° and a new joint value of -180°. The minimum change is clockwise +50°, whereas the maximum change is counterclockwise -310°. The minimum change is added to the previous joint value which may not lie in the range $\pm 130°$, compared to the joint limits. If acceptable, then this becomes the new joint value; otherwise, the maximum joint change is attempted and compared to the joint limits.
**WARNING!**: Some robot models have unique ways of treating revolute joint values by quadrant. When they differ from the CODE procedures, you must accommodate these differences in the device driver which the CIMServer uses to communicate motion commands to the existing robot controller. This will not be a problem, however, if the Cimetrix Open Architecture Controller is being used to drive the robot.

**Custom Exact Inverse Kinematics**

You can develop a custom invkin solution as follows:

- Create and compile a C language invkin solution routine.
- Give the routine an iclass number.
- Use that number to tell the CIMServer where to look for the solution.

Custom invkin can be based on the CIMServer’s DH parameters, although it is not recommended. If you use DH parameters, you may get them from the CIMServer or calculate your own as you wish. CODE routines provide the desired target pose of the last user joint frame relative to the base frame of the user robot as loaded into `rob->tdis`. This pose is calculated automatically for any tool that is currently attached to the robot and designated to move to a target pose in space which corresponds to some operational feature or pick/place pose.

If you use the CIMServer’s DH parameters to develop your custom invkin, you must pass the parameters `rob->parm`, `rob->rob_to_inv` and `rob->last_robjnt_to_invjnt` through your function argument in the include file `user_kin.CK`.

As explained on page 3-21, the `rob->parm` array is the input DH parameter values for the robot joints (numbered from 0 for the first joint, to `ndof-1` for the last joint). The homogeneous transformation that locates the user robot base frame relative to the DH invkin base frame is provided in `rob->rob_to_inv` in C row form. Note that the DH invkin base frame is the same as its first joint frame. The transformation describing the last or distal joint frame relative to the last DH invkin joint frame is provided in `rob->last_robjnt_to_invjnt`.

The invkin procedure now requires you to determine the target frame for the distal joint of the DH invkin robot. You must apply several functions from the matrix library to determine a new `tdis` target frame for the distal DH invkin joint in the DH invkin base frame (same as its first joint frame). The following matrix operations assume that the robot data structure is not passed as an argument to the actual function. In this function `new_tdis` must be declared as a matrix and is the new target:

```c
mul4x4(rob_to_inv, tdis, new_tdis); /*new_tdis declared as matrix*/
invmat(last_robjnt_to_invjnt, mat);
mul4x4(new_tdis, mat, new_tdis); /*new_tdis is desired target*/
```

As previously instructed, you simply enter your custom routine into the include file `user_kin.CK`. You should use `rob->nsoln` and `rob->soln` to store the custom solution as before. It might also help if you are familiar with the CIMServer robot data structure.
For the special case of inner TCFs as discussed in the section, “TCF’s Attached to Inner Joints” on page 3-24, the invkin procedures will have to be revised slightly for sensors and tools attached to interior joints. Normally the CIMServer passes in the \( \text{tdis} \) matrix which is the desired target frame for the last joint frame relative to the base frame of the robot. But in the case of a TCF attached to an inner joint, the \( \text{tdis} \) is now the target frame for the TCF and not for the joint which is the parent of the TCF.

In addition, the robot data structure parameter \( \text{tcf\_to\_joint} \) must be passed in as \( \text{robot\_tcf\_to\_joint} \) to locate the TCF relative to its parent inner joint. The user must then use \( \text{tcf\_to\_joint} \) and \( \text{tdis} \) to solve for the inner joint values.

The invkin procedures can be written to branch according to the joint to which they are attached, using the value of \( \text{inner\_tcf} \) which is added to the calling routine argument. If you use the CIMServer’s DH parameters to develop your custom invkin routine then the additional parameter \( \text{rob\_to\_inv} \) must be passed through as described previously and an additional parameter, \( \text{inner\_robjnt\_to\_invjnt} \), which is a member of the \text{command\_entry} data structure. The \( \text{tdis} \) must be modified in the inner TCF branch of the code to get \( \text{new\_tdis} \) and the \( \text{tcf\_to\_joint} \) must be modified by \( \text{inner\_robjnt\_to\_invjnt} \) to get \( \text{new\_tcf\_to\_joint} \). Both \( \text{new\_tdis} \) and \( \text{new\_tcf\_to\_joint} \) must be declared as matrices:

\[
\begin{align*}
\text{mul4x4(rob\_to\_inv, tdis, new\_tdis); /* new\_tdis of TCF */} \\
\text{mul4x4(inner\_robjnt\_to\_invjnt, tcf\_to\_joint, new\_tcf\_to\_joint);} \\
\end{align*}
\]

The API \text{CxGetInnerTcfOffset()} can be used to get the achieved target position and orientation offsets from the desired target. In many cases the reduced kinematics will only allow the robot to reach the origin of the desired target, but not the orientation. Refer to the \text{Programmer’s Reference Manual} for more details on how to use this API. Note that the user should set a blend policy of \text{CX\_MOVE\_WAIT} to get the target offsets using this API, since the target offsets can only be obtained when the move completes.

Suggested inner TCF implementation for custom inverse kinematics:

- Pass in additional parameters: \( \text{inner\_tcf}, \text{ndof}, \text{dofold}, \text{inner\_robjnt\_to\_invjnt}, \text{and tcf\_to\_joint} \).
- If the CIMServer’s DH parameters used for invkin, generate \( \text{new\_tdis} \) and \( \text{new\_tcf\_to\_joint} \).
- Branch invkin solutions by \( \text{inner\_tcf} \) value.
- Solve for reduced set of joint values.
- Set joint values of joints beyond \( \text{inner\_tcf to dofold values} \).
- In the CODE Application process, set the blend policy to \text{CX\_MOVE\_WAIT}.
- To get target offsets the call to \text{CxGetInnerTcfOffset()} should be placed immediately after the move command in the CODE application process.

You can also enter a custom numerical invkin solution (for robots that do not have an exact invkin) with this same interface. However, you should use the custom invkin only for a single custom robot, since generalization for a class of robots is more difficult because of the differences in zero states, joint polarity, joint ordering, TIFs, etc., even when the robots are of the same class.

To enter a custom invkin solution, follow these steps:

1. Load the workcell into CIMTools.
2. Open the Node Inspector.

3. Open the Browser, and click on the name of the robot node.

4. Update the Node Inspector for the robot node and get the Robot Panel.

5. Choose **Custom** from the **Inverse kinematics** option menu.

   A field will appear next to the **Inverse kinematics** option menu.

6. Type the iclass number of the custom inverse kinematics routine to be used in this workcell into the field that appeared.

An example of a simple custom invkin is shown in Code Listing 2-7. Although only a portion of this invkin routine is shown, and although the DH parameters are not passed through the argument list, this routine is a customization of a CIMServer DH invkin, and a new tdis is determined at the beginning of the routine.

```
/**
 * This is custom kinematics for IBM Glove Box robot. The robot is a regular 6-axis robot, plus a seventh translational axis (gripper).
 * joints are numbered as: 0, 1, 2, 3, 4, 5, and 6
 * matrix tdis = distal IK jnt rel to IK base
 * long *nsoln = number of valid invkin solns
 * double **soln = invkin solutions [nsoln][ndof]
 * long *soln_ok = solution OK (CX_TRUE\CX_FALSE) flag
 * double *dofold = last joint values
 * long ss = soln_selected
 * matrix rob_to_inv = usr base frm rel to IK base frm
 * matrix last_robjnt_to_invjnt = last user jnt rel to IK lst jnt

******************************************************************************
*/
#include <code/matrix.h>
#include <code/const.h>

void glovebox(matrix tdis, long *nsoln, double **soln, long *soln_ok, double *dofold, long ss, matrix rob_to_inv, matrix last_robjnt_to_invjnt)
{

  matrix ntdis; /* new tdis */
  matrix mat; /* tmp. matrix */
  matrix t6_inv; /* inverse matrix for the last joint */
  double d6;

  double c4,s4a,s4b,t35;

  /*calculate the new tdis*/
  mul4x4(rob_to_inv, tdis, ntdis);
  invmat(last_robjnt_to_invjnt, mat);
  mul4x4(ntdis, mat, ntdis);

  /* takes care of the gripper joint, in inverse mode, we only use
   * the last joint value for the gripper */

  d6 = dofold[6];
  identm( t6_inv );
  t6_inv[1][1] = 0.0;
```
t6_inv[1][2] = -1.0;
t6_inv[2][1] = 1.0;
t6_inv[2][2] = 0.0;
t6_inv[3][2] = -d6;
mul4x4(ntdis, t6_inv, ntdis);
soln[0][6] = soln[1][6] = dofold[6];

/* ntdis now represents the transformation from joint 5 to the
 robot base */
c4 = ntdis[2][0];
s4a = sqrtabs(1.0 - c4*c4);
soln_ok[0] = soln_ok[1] = CX_FALSE;

switch( ss ) {
    case 0: /* Solution #1 */
        /* joint 0 */
        soln[0][0] = ntdis[3][2];
        /* joint 1 */
        soln[0][1] = -ntdis[3][1];
        /* joint 2 */
        soln[0][2] = -ntdis[3][0];
        /* joint 4 */
        soln[0][4] = atan2(s4a, c4);
    /* joints 3 and 5 */
        if(fabs(fabs(c4)-1.0) > CX_EPSILONS ) {
            soln[0][3] = atan2(ntdis[2][2], ntdis[2][1]);
            soln[0][5] = atan2(ntdis[1][0], -ntdis[0][0]);
        } else {
            /* degenerate case, user can implement according to his needs */
            soln[0][3] = dofold[3]; /* previous dof value */
                if( c4 > 0.0 ) {
                t35 = atan2(-ntdis[1][1], ntdis[1][2]);
                soln[0][5] = t35 - soln[0][3];
            } else {
                t35 = atan2(ntdis[1][1], ntdis[1][2]);
                soln[0][5] = t35 + soln[0][3];
            }
        }
    soln_ok[0] = CX_TRUE;
    break;
.
.
Code Listing 2-7: Partial custom invkin for 7-axis gantry robot.

Numerical Inverse Kinematics

The CIMServer uses simple least squares routines based on the robot Jacobian to move the robot TCF
by forward kinematic iteration to its target pose. When the robot has more than six independent joints,
you can weight the relative importance of the joints in the routine redundant(), also contained in
the file user_kin.c, by appropriately scaling the Jacobian elements according to any user determined criteria.

Actually, you can weight the joints for robots having any number of degrees of freedom, although this may not be meaningful unless the weighting can simulate the actual control algorithms in the real robot.

Similar to the other kinematic interface routines, the numerical interface requires the user-developed scaling routine to be entered into the include file user_kin.NK which branches by rclass number to the particular user-developed routine for scaling the Jacobian. To enter the routine, follow these steps:

1. Load your custom robot workcell into CIMTools.
2. Open the Node Inspector.
3. Open the Browser, and click on the name of the robot node.
4. Update the Node Inspector to bring up the Robot Panel.
5. Choose **Numerical** from the Inverse kinematics option menu.

A field will appear next to the Inverse kinematics option menu.

6. Type the rclass number of the custom inverse kinematics routine to be used in this workcell into the field that appeared.

If the robot has more than six joints, there are an infinite number of joint solutions which can place a terminal control frame (TCF) representing an end-effector at some desired pose. Note also that some robots with 6 dof or less often have redundant configurations which occur when two or more joints become colinear.

**CODE Numerical Approach**

The numerical solution approach implemented in the CIMServer uses a least squares approach to determine inverse kinematics of a redundant robot. We define \( V = \frac{dp}{dt} \) = rate of change of pose vector (position plus orientation components) where \( p = [\text{roll pitch yaw x y z}] \) and the joint velocities are the derivatives of the joint values, \( q' = \frac{dq}{dt} \) so that in a numerical approximation, \( \Delta p = J \Delta q \) where \( J \) is the Jacobian.

A solution which minimizes the change in joint space follows. The equations in this section are shown in column form to be consistent with the literature expositions.

\[
\Delta q = J^T (J J^T)^{-1} \Delta p
\]

\[
\begin{array}{cccc}
\text{nx1} & \text{nx6} & \text{6x6} & \text{6x1}
\end{array}
\]

Given an initial TCF and a target TCF we apply the screw equations to determine \( \Delta p \) in the robot base frame, where \( \Delta p = [\theta_x \theta_y \theta_z x y z] \) and where \( \theta_x \theta_y \theta_z \) are the components of the screw rotation which is determined from the screw angle \( \theta \) and screw vector \( e \) by \( \theta_x = \theta e_x, \theta_y = \theta e_y, \theta_z = \theta e_z \).

\( e_x, e_y, e_z \) are the direction cosines of the screw vector and \( x, y, z \) are the translational components.

Given \( \Delta p \), we can estimate the necessary change in joint values by

\[ q_n = q_l + K \Delta p \]
where \( n = \text{next} \) and \( l = \text{last} \) and where \( K = J^t (J J^t)^{-1} \). The procedure continues to make joint changes until the initial TCF is at the target TCF to within a convergence criteria.

The above equations do not work well in some situations when the joint types are mixed, or the joint links have widely differing dimensions, because the terms in \( K \) become ill-conditioned. To account for this, the routine `redundant()` provides an interface for you to scale joint values and the terms in the Jacobian \((J_{ij})\), according to the following equations:

\[
\begin{align*}
    s_i &= a_i q_i \\
    G_{ij} &= J_{ij} / a_i
\end{align*}
\]

where the \( a_i \) are the scale factors, \( q_i \) the joint values, \( s_i \) the scaled joint values, and \( G_{ij} \) the scaled Jacobian elements.

The function of the user-developed routine is to provide the scale factors and enter them into the vector parameter `scale`. The choice of the scaling factors can be chosen according to any suitable criteria, and provides flexibility for managing the inverse kinematics of robots such as a robot attached to a large gantry.

Most of the scaling difficulty comes when robots mix revolute and translational joints. For example, consider a robot which has a first revolute joint reach of 2000 mm and other joints that are translational. A first joint change of 1 radian would rotate the robot gripper through a distance of 2000 mm, while a unit joint change in a translational joint would move the gripper by only 1 mm.

This illustrates the ill-conditioning spoken of. To avoid this problem, the scale factors \( a_i \) for the translational joints should be chosen such that the scaled Jacobian components for the translational joints have values near those of the Jacobian components for the revolute joints. This requires that the weighting of the joints be such that the value of the translational joint be decreased proportional to the ill-conditioning magnitude.

For this example, the \( a_i \) values for the translational joints could be \( 1/2000 \). This would effectively increase the magnitude of the translational Jacobian components to that of the revolute Jacobian components, eliminating the ill-conditioning.

The routine `redundant()` which follows branches by `rclass` number to your routine in the include file `user_kin.NK`. Note that in your routine you should only declare `scale` as a double pointer.

Scaling values must be specified for each independent joint (`rob->ndof` = number of independent joints).

```c
/
*--------------------------------------------------redundant--------------------------------------------------*
REMARKS
this routine provides a user interface for user defined functions
that specify the Jacobian scaling factors for redundant robots;
the rclass = 0 case weights the scaling factors based on their
joint range limits
------------------------------------------------------------------*/

void redundant(robot *rob, double *scale)
{
    long i;
    long rclass; /* positive value of rob->rclass */
    
    /* switch only on positive rclass number but allow negative
    rob->rclass number to select numerical invkin least
    squares solution normally applied to ndof > 6 robot for redundant
    6-axis robot */
    rclass = labs(rob->rclass);
}
```

Customizing CODE
switch(rclass) {
    case 0: /* default case */
    for(i=0;i<rob->ndof;i++)
        scale[i] = 1.0;
    break;

    #include "user_kin.NK"

    default:
    rob->rclass = 0;
    for(i=0;i<rob->ndof;i++)
        scale[i] = 1.0;
    break;
}
} /* end of redundant() */

Code Listing 2-8: The routine for specifying joint weightings for numerical invkin.

The user_kin.NK include file reserves Case 0 for the weighting of the independent joints based on their joint ranges. Note that a large scale value will minimize the participation of a joint in the invkin solution.

Since the scaling routines are called in each motion iteration, you could dynamically change the scale values during a simulated task, thereby modifying the relative importance of the joints based on proximity to targets or other rationale.

    case 1000: user_routine(rob,scale); break;

Code Listing 2-9: Sample include file user_kin.NK

NOTE: For the special case of a 6-axis robot that is redundant in many configurations and for which you have no exact invkin, you can enter a negative rclass number to make the numerical invkin choose a set of joint values which minimize the joint change from one configuration to the next. Otherwise the numerical invkin procedures will probably fail.

Numerical invkin procedures should be treated carefully and not be expected to represent a commercial robot’s actual performance. Numerical procedures should be used only in absence of an exact invkin for the robot for the purpose of simulating a robot’s kinematic capabilities.

A number of problems can arise while applying a numerical invkin to simulate tasks:

1. The robot numerical convergence procedures may choose entirely different joint configurations when moving between separated targets because the gradient techniques may not be able to distinguish between the available solutions.
   If you reduce the interpolation size in path following tasks, this will not happen because each interpolation step will place the robot near the desired robot joint configuration in the next step.

2. When moving between discrete configurations, you must usually start in a joint configuration that is near the one desired. Otherwise, the invkin procedures may converge to a solution that is not desired, causing the robot to “flip” between configuration types.
3. Alternatively, you can set scaling values that weight the importance of joint changes differently and literally force certain joints to behave as desired. Of course, the real robot may not be able to emulate these motion characteristics.

**Forward Kinematics and Functional Dependencies**

The forward kinematics interface routine `func_dep_jnt()` (found in the file `user_kin.c`) is used to program nonlinear joint dependencies. This routine is used when nonlinear (functional) joint dependencies exist between dependent and independent joints such as those found in a four-bar linkage used to configure an independent robot link. `func_dep_jnt()` can also be used to specify variable joint limits for robots which have their joint limits vary with robot configuration.

Like the `invkin` routines, the routine `func_dep_jnt()` is called at every trajectory step to define dependent joint positions and/or limits. This routine is called by the CIMServer’s forward kinematics. Shown below is the interface routine `func_dep_jnt()`.

```c
/*-----------------func_dep_jnt-------------------------------------
REMARKS
  func_dep_jnt branches by rob->fclass number to user defined function for
determining joint limits, joint values, etc.
rob: pointer to the robot structure
jnt_to_check: specifies which independent joint moved
  (CX_CHECK_ALL for all)
set_to_limit: if CX_TRUE adjust the joint value to limit
return: programmer must return CX_FALSE if joints
  limit exceeded, else CX_TRUE
----------------------------------------------------------------*/
long func_dep_jnt(command_entry *q,long jnt_to_check, long set_to_limit)
{
  robot *rob;

  rob = q->rob;

  if ( ! rob )
    {
      robline_errno = CX_ROB_NOT_FOUND;
      return ( CX_ERROR ) ;
    }

  switch(rob->fclass) {
    #include "user_kin.FK"
    
    default:
      robline_errno = CX_NO_DEP_JNTS_ROUTINE;
      break;
  }

  return(CX_ERROR); /* only reached for default case
(see user_kin.FK) */

ten *func_dep_jnt */
```
The arguments which you should use in the developed routine are listed as follows:

- **q**: Pointer to a command_entry data structure for current move (current_command). The command_entry data structure is defined in `<code/server.h>`.
- **rob**: Pointer to the robot data structure (defined in `<code/robot.h>`).
- **jnt_to_check**: Input number which specifies which independent joint moved. If jnt_to_check = CX_CHECK_ALL, then your routine must check all independent joint limits.
- **set_to_limit**: If specified as CX_TRUE, then the routine must set the joint’s value to its limit if that joint’s limit is exceeded.

The routine must return **TRUE** if the solution falls within legal joint limits and **FALSE** if any of the joint limits are exceeded. Note that if the joint limits are exceeded, the routine should return **CX_FALSE**, even if the **set_to_limit** flag is **CX_TRUE**.

Other key data structure parameters are as follows:

- **q->dof[i]**: Joint value for joint i (in radians if CX_ROTATE)
- **rob->ndof**: Number of independent joints
- **rob->njoints**: Number of all joints, including dependent joints
- **rob->jtype**: Joint type (either CX_ROTATE or CX_TRANS)
- **rob->dofold[i]**: Joint value i at last robot configuration
- **rob->joints[i] ->dofmin**: Minimum joint limit for independent joint i
- **rob->joints[i] ->dofmax**: Maximum joint limit for independent joint

**WARNING!**: Do not modify the values of **rob->ndof**, **rob->njoints**, and **rob->jtype** in the kinematics routines provided.

Once you generate the functionally dependent routine, it can be entered into the include file `user_kin.FK` as a C language case switch statement as shown in the code listing 2-11. The `user_kin.FK` include file is found in the directory `$ROBTOP/lib/cimetrix/custom` and is user-modifiable.

**Fclass** numbers should be similar to the **iclass** number for the robot. The CIMServer expects these routines to return a **FALSE** if the joint limits are exceeded, otherwise, the routine should return a **TRUE**.
You can specify any \textit{fclass} number not equal to 0 in the CIMServer. The \textit{fclass} number is used to switch to the appropriate routine in \texttt{func\_dep\_jnt()} to determine dependent joint values for complex robots and/or to determine joint limits when they vary as a function of robot configuration. To specify the \textit{fclass} number for your robot, follow these steps:

1. Load the custom robot workcell into CIMTools.
2. Open the Node Inspector.
3. Open the Browser, and click on the name of the robot node.
4. Update the Node Inspector to get the Robot Panel.
5. In the pop-up menu under \texttt{Control}, select \texttt{Forward Kinematics}.
6. From the buttons on the right, click on \texttt{Function}.
7. Type the \textit{fclass} number of the functional dependency routine to be used in this workcell into the field that appears.

An example of a forward kinematics routine is listed in Code Listing 2-12. Note two routines that the user could call, \texttt{check\_jnt\_limits()} for verifying that joint limits have not been exceeded, and \texttt{load\_jnt\_matrix()} for updating the joint matrix with the new joint value specified in the routine. If \texttt{joint\_to\_check} is -1 then all joints are checked against their limits, else only the entered joint number is checked. If the argument \texttt{set\_to\_limit} is set to \texttt{CX\_TRUE}, then any joint exceeding its limit will be set to the limit, else \texttt{check\_jnt\_limits()} will return \texttt{CX\_FALSE} if a joint exceeds its limit.
double x, y, ang;

/* first check to see if independent joints are within legal limits */

possible = check_jnt_limits(q, jnt_to_check, set_to_limit);
if(!possible && !set_to_limit) return(CX_FALSE);

/* now calculate values for functional dependent joints */

if( jnt_to_check == -1 ) {
    q->dof[6] = q->dof[7] = 0.0;
    load_jnt_matrix(q, 6);
    load_jnt_matrix(q, 7);
} else if(jnt_to_check == 1) {
    ang = 2.318 + q->dof[1];
    x = 348.93 + 223.61*cos(ang);
    y = 223.61*sin(ang);
    q->dof[7] = atan2(y, x) - 0.6944;

    ang = 0.708 - q->dof[2] - q->dof[1];
    y = 208.0*sin(ang);
    x = 348.93 - 208.0*cos(ang);
    q->dof[6] = atan2(y, x) - 0.617;

    check_jnt_limits(q, 2, CX_TRUE);
    load_jnt_matrix(q, 2);
    load_jnt_matrix(q, 6);
    load_jnt_matrix(q, 7);
} else if (jnt_to_check == 2) {
    ang = 0.708 - q->dof[2] - q->dof[1];
    y = 208.0*sin(ang);
    x = 348.93 - 208.0*cos(ang);
    q->dof[6] = atan2(y, x) - 0.617;

    load_jnt_matrix(q, 7);
}

return(possible);
}

Code Listing 2-12: Forward kinematics for a Panasonic robot.

Branching Robots

Automating the CIMServer inverse kinematics for branching robots such as machine tools (CX_NC_ROBOT) is more complex kinematically than doing the same thing for the serial robot (CX_SERIAL_ROBOT), as shown in Figure 2-3 below. The CIMServer’s auto DH methods can handle robots with two simple branches growing from the robot base if the user correctly constructs the user CX_NC_ROBOT. Without a detailed description, note that the CIMServer artificially treats a simple branching robot as a serial robot in the relative motion sense by treating the terminal link j as the base and renumbering the joints in a serial fashion from joint j to joint n-1, where joint j now becomes the
first joint, joint j-1 becomes the second joint, etc., and where n is the robot’s total number of degrees of freedom.

You do not need to be concerned with most of these details. The CIMServer internally accounts for the variable base and targets that are attached to the base branch, once the user constructs the robot tree according to the rules described shortly. Link j is the terminal link of the CX_NC_ROBOT base branch. The base branch is the branch to which the part or target is attached. The remaining branch is referred to as the tool branch and requires that the tool (probe, bit, etc.) be attached to link n-1 (the terminal link attached to joint n).

**Constructing the Branching Robot**

The CIMServer’s auto DH methods will work if the you use the following as guidelines when constructing robot:

1. Construct the CX_NC_ROBOT using normal CIMServer procedures, ensuring that the base branch joints are numbered from 0 to joint j and the tool branch joints are numbered from j+1 to n-1, where j < n-1 and n is 6 or less.
2. Check the joint panel for each joint to ensure that the CIMServer internal joint numbers are ordered as necessary (joint number automatically assigned by the CIMServer). If the base or tool branch joints are ordered incorrectly, cut the branch and copy it to the robot base frame to correct the order.
3. Attach all target elements to the terminal joint/link of the base frame (joint j). This means that the target will move as joints 0 through j move.
4. Identify only tool TCF’s that are attached to the terminal joint/link (joint n-1).

The CIMServer will alert you if any of the guidelines listed above are not followed correctly.

**Aligning Tools to the Z Axis**

The CIMServer’s auto DH methods also require Z axis alignment for CX_NC_ROBOTs that have orientation joints. Typically four and five-axis CX_NC_ROBOTs will have one or two rotational joints. In such cases you must orient the TCF so that its Z axis is perpendicular to the last rotational joint axis of the robot.

Similarly, the part target frame(s) for any part attached to the base branch must be aligned to ensure that the TCF Z axis can be aligned with the target Z axis. For example, if a four-axis machine tool is machining a surface, the path frames should have their segment frames oriented such that the Z axis is normal to the surface. Any generative process which generates tool paths in terms of the CIMServer’s curve segments should ensure Z orientation in this way.

Three-axis CX_NC_ROBOTs with only translational joints do not require Z axis alignment if the user sets the tool motion type to CX_FIXED_ORIENT using the API CxSetToolMotionType().
**WARNING!**: The CIMServer’s ordering scheme may not match the actual joint numbering schemes used by a particular model of a machine tool. If CIMServer motion control is used to drive the CX_NC_ROBOT through its existing controller, then you may have to account for the differences in the device driver that communicates CIMServer motion to the CX_NC_ROBOT controller. This will not be a problem if the Cimetrix Open Architecture Controller is being used to drive the machine tool.
CHAPTER 4

Mechanism & I/O Interfacing

The open architecture of the CIMServer provides software “hooks” whereby users can integrate custom software interfaces to mechanism and I/O controllers. This chapter discusses the procedural steps in developing custom controller interfaces, and contains seven major sections:

- “Controller Interfacing Architecture”
- “Developing Controller Interfaces”
- “Important Data Structures”
- “Controller Interface Functions”
- “Handling Driver Errors”
- “Adding Driver Modules and Libraries to the Makefile”
- “Example”

The first section, “Controller Interfacing Architecture,” discusses the methodology which allows the CIMServer to interface directly with hardware devices. “Developing Controller Interfaces” provides an overview of the functions you need to develop. The “Important Data Structures” section discusses the data structures pertinent to developing controller interfaces. “Controller Interface Functions” provides a detailed functional specification for the functions required. The “Adding Driver Modules and Libraries to the Makefile” section describes how to integrate your functions with the CIMServer. The “Example” section provides source code for a sample controller interface.

Controller Interfacing Architecture

Symmetric Interface

CODE incorporates a symmetric interface to plan and control workcell activities. Under this architecture, you can obtain graphical verification of workcell tasks using off-line simulation, then execute the same command or group of commands on-line. The same application program which drives the simulation also drives the workcell hardware devices.

The CIMServer provides open, dynamic real-time access to a hierarchical database of the workcell objects (the Cell Model). It also incorporates an interface to a signal table, allowing real-time access to either hardware or software I/O signals.
**Client-Server Interaction**

Task-level CODE application programs are developed in the C programming language using the CODE Application Programming Interface libraries (API libraries). The CODE API libraries contain C functions which define an Application Programming Interface (API) to the CIMServer. The CODE API functions communicate with the CIMServer through the CODE Messaging Protocol. The CIMServer uses these messages to manage objects within the hierarchical database of the workcell and, optionally, to drive physical hardware components, including mechanisms and other devices interfaced to the system through I/O signals. This architecture is illustrated in Figure 3-1. This feature allows all cell devices to be programmed using a uniform programming interface.

When the CIMServer receives a message from an application process, it channels the command to a controller-specific interface function, directly linked into the CIMServer executable file. These driver functions use information and data from the CIMServer’s workcell model (including the Cell Model and signal table) to interface with the device’s host controller. Under this architecture, the process logic resides in the application program. Only a rudimentary set of commands are used to drive the actual mechanisms or I/O signals through their native controllers. This feature allows you to easily develop CODE device drivers and eliminates the need to support complete language translators since the device controllers require only an elemental set of commands to complete device moves or I/O signal interactions.

**Driver Implementation Methods**

This architecture provides a very flexible and open way to integrate custom device controller interfaces. You can implement network based communication links, serial communication links, or bus level device drivers to interface with a large variety of control interfaces. The system is limited only by the capabilities of the specific controller interface you intend to use. Under this architecture, the functions developed to interface with a device controller are responsible for the following tasks:

- **Command Interpretation**: The functions must take data from the CODE Cell Model or signal table, and format a command to be sent to the controller for execution. This may involve converting data into a format and unit that the controller can understand. For example, the CIMServer internally represents angular data in radians. Many mechanism controllers require angular information in degrees. The interface functions must perform the radians to degrees conversion.

- **Communication Protocol**: The interface to the device controller must also handle any data communication required to interact with the controller. The nature of this interface is a function of the capabilities of the device controller. The protocol interface must ensure a robust link between the device controller and the CIMServer, and is responsible for performing handshaking and data integrity checking.
Any errors occurring on the device controller must be reported back to the CIMServer interface. These errors are ultimately reported back to the application process. In addition, the interface functions should be designed in such a manner that error recovery is possible without having to manually re-initialize the system.

The CIMServer supports two independent types of controller interfaces. These include:

- Mechanism controllers
- I/O signal controllers

The controller interfacing architecture is implemented by providing the user with a set of functional connection points (or hooks) into the CIMServer. These hooks define places in the code where a user’s controller specific function will be called when the Server attempts to perform a specific task, such as commanding a move, setting an I/O signal, etc. These connection points are defined as “case switches”, which switch on an integer constant defining the controller’s type. An include file is defined for each functional connection point.

In the following sections, the specific functions required to interface a mechanism or I/O controller with the CIMServer are reviewed. The information presented is divided into six main sections. These include:

- “Developing Controller Interfaces”
- “Important Data Structures”
- “Controller Interface Functions”
- “Handling Driver Errors”
- “Adding Driver Modules and Libraries to the Makefile”
- “Example”

The “Developing Controller Interfaces” section provides a brief overview of the interface functions. It also provides a brief overview of different versions of the CIMServer, and discusses how the different versions affect controller interface development.

The “Important Data Structures” section provides an in-depth description of the data structures available to the mechanism and I/O controller interface functions.

The “Controller Interface Functions” section provides a reference of each function and includes a detailed functional specification, defines the corresponding include file into which the function must be integrated, information about when each function is called, required return values, available information, etc.

The “Handling Driver Errors” section provides detailed information about how to handle errors in your controller interface.

The “Adding Driver Modules and Libraries to the Makefile” section describes how the user’s functions and any associated libraries can be linked with the CIMServer.

The “Example” section provides a sample interface module to the Cimetrix OAC library, which interfaces to the Delta Tau PMAC card.
Developing Controller Interfaces

As described in the previous section, the mechanism and I/O interface functions are linked directly with the CIMServer. To develop an interface to a particular controller, the first step is to determine the capabilities of the target controller and which driver interface functions are required. These capabilities are integrated into five functions which are described as follows:

- **xxxx_motion_parm**: This function is used to implement motion controller capabilities used to get/set motion control parameters.
- **xxxx_motion_func**: This function is used to implement motion control functions.
- **xxxx_signal_func**: This function is used to implement capabilities related to specific I/O signals.
- **xxxx_signal_drvr_func**: This function is used to implement capabilities related to I/O signal interface drivers.

![Figure 3-1: CODE System Architecture.](image-url)
**xxxx_send_cmd** This function is used to send controller specific commands to the controller.

NOTE: When you develop these functions, the *xxxx* will be replaced with your own valid prefix (e.g. *my_motion_parm*). Once developed, these functions can be linked directly with the CIMServer. These functions are integrated into the CIMServer in the file *user_cntrl.c* in the directory `$ROBTOP/lib/cimetrix/custom`, where `$ROBTOP` is an environment variable defining the installation directory.

The user integrates their controller specific functions into a look-up table, found in the *user_cntrl.c* module. Once the required functions have been developed and integrated, the user then compiles and re-links a new CIMServer executable file.

Detailed instructions on the required functionality, and arguments to each function are given in “Controller Interface Functions” on page 4-21. Instructions on how to rebuild the CIMServer executable are found in “Adding Driver Modules and Libraries to the Makefile” on page 4-40.

### Threaded vs. Non-Threaded Versions of the CIMServer

There are two basic versions of the CIMServer, a threaded version and a non-threaded version. In the threaded version, specific threads, acting as separate processes, are started when the server starts or when a mechanism is opened. These threads monitor I/O signal changes and controller error conditions, process motion requests, communicate with client processes, etc. In the threaded version, many of these threads can block (i.e., go to sleep), waiting for an event to occur while other threads associated with the same process continue uninterrupted processing.

In the non-threaded version, because there is no parallel processing within the context of a single process, functions cannot block; therefore, they must be polled. For example, in the threaded version, a thread can block waiting for the end of a motion to occur, whereas in the non-threaded version, to be able to continue to process incoming events, the system cannot block for an end of motion condition. The system must poll for the end of motion condition.

### Important Data Structures

This section contains a description of six important data structures. These data structures are defined as follows:

- `signals_t`
- `signal_drivers_t`
- `mechanism_entry`
- `command_entry`
- `cmd_msg`
- `CxController`

These data structures provide all essential information required to interface to mechanism and I/O controllers.
The signals_t Data Structure

The signals_t structure is used to define an entry of a specific I/O signal in the signal table. Much of the information contained in this data structure is defined in the CODE signal table. This data structure is defined as follows:

NOTE: Fields marked read only should not be changed.

```c
/* data on physical signal instances */
typedef struct signals_t {
    char name[NAME_LEN]; /* ASCII name of logical signal */
    long input; /* hardware Input */
    long output; /* hardware Output */
    long use_init; /* Use init_value to initialize signal */
    long init_value; /* value to set on startup */
    long Field1; /* Driver specific field 1 */
    long Field2; /* Driver specific field 2 */
    long Field3; /* Driver specific field 3 */
    long Field4; /* Driver specific field 4 */
    long Field5; /* Driver specific field 5 */
    long Field6; /* Driver specific field 6 */
    long Field7; /* Driver specific field 7 */
    long Field8; /* Driver specific field 8 */
    long Field9; /* Driver specific field 9 */
    long Field10; /* Driver specific field 10 */
    long value; /* current signal value */
    long sim; /* simulation toggle */
    caddr_t sigInfo; /* generic pointer drvr specific sig str */
    signal_drivers_t *sdrvr; /* Pointer corresponding signal driver data structure. */
    struct table_entry *first_entry; /* Pointer for internal CIMServer use */
    long table_index; /* Index into signal table array, for internal CIMServer use */
    char *expression; /* Character string used to implement logical expressions */
} signals_t;
```
char *description; /* Character string describing use of the signal */
char *sig_class; /* Character string defining the signal class */
}
signals_t;

The fields in this data structure are defined as follows:

**input**
Defines whether the signal is available as a hardware input. A value of 0 indicates that the signal is not readable in hardware. A value of 1 means the signal must be polled. A value of 2 means that the corresponding driver can generate interrupts for this signal, and does not require polling. (READ ONLY).

**output**
Defines whether the signal is available as a hardware output. A value of 0 indicates the signal does not have output capability. A value of 1 indicates that it does. (READ ONLY).

**use_init**
Only used when the output field is 1, and is a flag indicating whether or not to initialize the signal’s state when the CIMServer is started (1), or to leave the signal in its present condition (0). (READ ONLY).

**init_value**
Only used when the output field is 1 and the use_init field is 1. When these conditions are satisfied, the init_value defines the desired initial value of the signal when the CIMServer starts. (READ ONLY).

**Field1 - Field10**
These fields correspond to driver definable parameters specified in the signal table. These fields are generally used to define the hardware signal mapping. For example, the signal may correspond to a specific signal on the I/O controller. These fields would be used to define how to access the specific signal on the I/O controller. (READ ONLY).

**value**
Defines the current value of the signal. (READ ONLY).

**sim**
Defines whether the signal is in CX_ANIMATION or CX_RUNTIME mode. (READ ONLY).

**sig_info**
Generic pointer to user-definable data. This field can be used by the user to store other driver information specific to a signal. (READ WRITE).

**sdrv**
A pointer to a signal_drivers_t data structure which corresponds to the controller associated with this signal. (READ ONLY).

**table_index**
Defines the numerical index into an internal CIMServer array of signals (READ ONLY).
expression  A character string defining a logical expression. This field can be used by the signal driver to define a logical condition which must be CX_TRUE for the signal to return CX_TRUE (READ ONLY).

description  A character string providing a description of the I/O signal’s use (READ ONLY).

sig_class  A character string defining a classification to which the I/O signal belongs (READ ONLY).

The signal_drivers_t Data Structure

The signal_drivers_t structure is used to define I/O signal driver parameters. This data structure is defined as follows:

```c
typedef struct signal_drivers_t { /* Signal device driver struct */
    long driver_type ; /* Type of device driver. */
    long driver_instance; /* Which instance of the driver. */
    char device_name[PATH_LN]; /* Logical device name. */
    long start_index ; /* Starting signal index. */
    long end_index ; /* Ending signal index. */
    long poll_flag ; /* T/F indicating polled signals */
    long intr_flag ; /* T/F indicating intr signals */
    char labels[10][NAME_LEN]; /* Optional signal parm labels */
    caddr_t drvrInfo; /* Generic ptr to drvr specific info */
    long initialized; /* T/F indicating driver inited. */
    long num_polled_table_entries ;/* Number of polled signals associated with the driver */
} signal_drivers_t;
```

The parameters defined in this data structure are defined in the signal table. The fields of this data structure are defined as follows:
driver_type  
A parameter which defines the signal controller type. The value placed in this field should be defined in the $ROBOT/include/code/cntr_const.h file (where $ROBOT defines the installation directory). (READ ONLY).

driver_instance  
Defines an instance of a specific driver. For example, you may be connected to several different controllers of the same type. This field is used to distinguish between them. (READ ONLY).

device_name  
This field defines a logical device name used to communicate with the device (e.g. /dev/com1 if the controller is connected to the system through serial port 1). (READ ONLY).

start_index  
Defines the starting index into the signal table for signals corresponding to this driver. (READ ONLY).

end_index  
Defines the ending index into the signal table for signals corresponding to this driver. (READ ONLY).

poll_flag  
Indicates instances of polled signals within this driver (1 - True, 0 - False). (READ ONLY).

labels  
Character strings defining labels used when CODE creates a signal table. (READ ONLY).

drivrInfo  
A pointer to user defined data containing information specific to an I/O controller. (READ WRITE).

initialized  
Used to indicate initialization status of an I/O controller. (1 - Initialized, 0 - Not initialized) (READ WRITE).

num_polled_entries  
Used internally by CODE. (READ ONLY).

poll_state_changed  
Used internally by CODE. (READ ONLY).

signal_poll_check  
Used internally by CODE. (READ ONLY).

poll_state  
Used internally by CODE. (READ ONLY).

The signal_info Data Structure

The signal_info data structure contains information for specific cntrl_signal_func functions. The data structure is defined as follows:

typedef struct signal_info {

mechanism_entry *mech ;

signals_t *sigPtr ;

long comparison ;

long value ;

long operation;

long change ;

} signal_info ;

These fields are explained in the description of the cntrl_signal_func function (see “Implementing the xxxx_signal_func Function” on page 4-33).

The mechanism_entry Data Structure

The mechanism_entry data structure contains information associated with a mechanism. The definition of this structure is found in the header file $ROBTOP/include/code/server.h

The main fields of interest in this data structure for developing mechanism controller interfaces include the following:

caddr_t cntrl_info;

long controller_type;

long controller_stat;

long controller_attr;

long open_mode;

long err_no;

These fields are described as follows:
cntrl_info  Defines a generic pointer to user definable data in which information specific to a mechanism controller can be stored and accessed from the CIMServer’s data base. This data structure is initialized in the cntrl_open function. (READ WRITE).

cntrl_type  Set using the API CxSetControllerType in a CODE application process. This flag defines the type of control interface for this mechanism. The value placed in this field should be defined in the $ROBTOP/include/code/cntr_const.h file (where $ROBTOP defines the installation directory). (READ ONLY).

cntrl_stat  A bitmask used to determine the current operating status of the controller. The possible mask values are defined in robconst.h, and include the following:
    CX_CONTROLLER_INITIALIZED
    CX_AMPS_ENABLED
    CX_MECHANISM_HOMED
    CX_TEACH_PENDANT_ACTIVE
    CX_KEY_SWITCH_ACTIVE
    CX_MECHANISM_HALTED
This mask is defined when the cntrl_get_status function is called. The user can get the status flags using the API CxGetControllerStatus in a CODE application process. (READ WRITE).

cntrl_attr  A bitmask used to define the trajectory generation capabilities of the servo control interface. The possible mask values can be found in the robconst.h header file, and include the following:
    CX_TRAJECTORY_MODE
    CX_CNTRL_TELE_MODE
    CX_INVKIN_MODE
    CX_FWDKIN_TIME_MODE
    CX_FWDKIN_RATE_MODE
    CX_TEACH_PENDANT_MODE
    CX_INVKIN_BLEND_MODE
    CX_FWDKIN_BLEND_MODE
The user can determine the controller attributes by calling the API CxGetControllerAttr in a CODE application process. This mask is set when the CX_CNTRL_GET_CONTROLLERATTRIBUTES function is called. (READ WRITE).

open_mode  A bitmask defining how the mechanism was opened. Possible mask values include:
CX_CONTROL - Mechanism is open, and a CODE application process has control of the mechanism.

CX_MONITOR - Mechanism is open, and is monitored by a CODE application process

This flag is useful in ensuring that the mechanism controller is currently only being accessed for CX_CONTROL by a single process. (READ ONLY).

err_no A parameter used to store a constant indicating the cause of a mechanism control error. The value of this constant is reported back to the application process. (READ WRITE).

The command_entry Data Structure

The command_entry data structure is used to queue up functions requiring sequential interaction with a servo controller. The definition of this data structure can be found in the header file $ROBTOP/include/code/server.h. The command_entry structure contains all information required for the completion of moves. The pertinent fields are defined as follows:

long type;
long start_signal;
long start_value;
long stop_signal;
long stop_value;
tree_node *tcf;
tree_node *target;
long target_same_as_tool;
matrix tcf_initial;
matrix tcf_final;
matrix tcf_target;
matrix tool_offset;
long tool_is_offset;
long tool_motion_type;
matrix fix_frame;
long soln_selected;
long trajectory_mode;
long sim;
double *dof;
double *dof_act;
double tip_set;
double screw_set;
double jnt_set;
double *curspd;
double *curspd_act;
long accel_flg;
double ramp_time_rise;

double ramp_time_fall;

double accel_rise;

double accel_fall;

double screw_accel_rise;

double screw_accel_fall;

double rise_S_time;

double fall_S_time;

long interp_type;

matrix mid_arc;

These fields are described as follows:
type  Defines the type of queue element. Possible types can include:
    CX_ROB_MOVE - a commanded move.
    CX_ROB_HOME - home the mechanism.
    CX_ROB_TEACH_PENDANT - invoke the controller's teach pendant.
    CX_ROB_WAIT - Wait for all buffered moves to complete.
    CX_ROB_UPDATE_JNTS - Update the current joint positions after all buffered moves are complete.
    (READ ONLY)

start_signal  Defines the CODE signal table index for a signal to be triggered when the motion defined in this command_entry starts. If no start signal is desired, this parameter is set to -1. The Server takes care of setting the start signal when required. (READ ONLY)

start_value  Defines the desired value of the start signal when motion associated with this command starts. (READ ONLY)

stop_signal  Defines the CODE signal table index for a signal to be triggered when the motion defined in this command_entry reaches its commanded position. If this signal is set to -1, no stop signal is desired. The Server takes care of setting the stop signal when required. (READ ONLY)

stop_value  Defines the desired value of the stop signal when motion associated with this command reaches its final target position. (READ ONLY)

tcf  Pointer to a CIMServer tree_node data structure which defines the node associated with the currently defined TCF. (Terminal (or Tool) Control Frame). (READ ONLY)

target  Pointer to CIMServer tree_node data structure which defines the node associated with the currently defined Cartesian target node. (READ ONLY)

target_same_as_tool  If this parameter is CX_TRUE, the move is defined relative to the currently defined TCF. (READ ONLY)
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>(READ ONLY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcf_initial</td>
<td>A matrix defining the current pose (position and orientation) of the TCF tree node relative to the “world” tree node.</td>
<td></td>
</tr>
<tr>
<td>tcf_final</td>
<td>A matrix defining the final pose of the TCF tree node relative to the “world” tree node.</td>
<td></td>
</tr>
<tr>
<td>tcf_target</td>
<td>A matrix defining the target pose of the TCF tree_node relative to the “world” tree node.</td>
<td></td>
</tr>
<tr>
<td>tool_offset</td>
<td>A matrix defining the tool offset. This matrix defines the offset from the target frame, and is applied during motions commanded relative to a target.</td>
<td></td>
</tr>
<tr>
<td>tool_is_offset</td>
<td>A flag indicating whether or not the tool_offset matrix is an identity matrix (CX_TRUE or CX_FALSE).</td>
<td></td>
</tr>
<tr>
<td>tool_motion_type</td>
<td>A parameter which defines the tool motion type. The possible values include:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_FULL_POSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_FIXED_ORIENT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_Z_POSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_Z_POSE_NO_SPIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_X_TANGENT_POSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>These parameters are described in Chapter 3, “Motion Control”, of the CODE Applications Programming Manual.</td>
<td></td>
</tr>
<tr>
<td>fix_frame</td>
<td>A matrix defining fixed orientation of the tool relative to the “world” tree node.</td>
<td></td>
</tr>
<tr>
<td>soln_selected</td>
<td>Defines the inverse kinematics solution selected in calculating target joint values.</td>
<td></td>
</tr>
<tr>
<td>trajectory_mode</td>
<td>A constant defining the trajectory mode. The constant can be defined as one of the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_END_POINT_MOVE - The controller will perform trajectory generation. Only target information is sent to the controller.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CX_CALCULATE_TRAJECTORY - The CIMServer will perform trajectory generation, and send trajectory information to controller.</td>
<td></td>
</tr>
</tbody>
</table>
sim A flag defining if the current move is to be done in CX_ANIMATION or CX_RUNTIME. (READ ONLY)

dof An array of values defining the target joint values of the independent joints associated with a mechanism. (READ ONLY)

dof_act An array of values defining the target joint values of the actuators associated with the independent joints of a mechanism. (READ ONLY)

tip_set The currently defined desired tip speed for non joint interpolated moves. (READ ONLY)

screw_set The currently defined desired screw speed for non joint interpolated moves. (READ ONLY)

jnt_set The currently defined desired joint speed for joint interpolated moves. This parameter has a range of 0 to 1. (READ ONLY)

curspd An array of values defining the current desired speed of each of the independent joints associated with a mechanism. (READ ONLY)

curspd_act An array of values defining the current desired speed of the actuators associated with the independent joints of a mechanism. (READ ONLY)

accel_flg A flag defining what type of acceleration parameter to use. The possible values of this flag include:
CX_CONST_RAMP_TIME
CX_CONST_RAMP_ACCEL
(READ ONLY)

ramp_time_rise Defines the acceleration ramp time. This parameter is used if the accel_flg is set to CX_CONST_RAMP_TIME. (READ ONLY)
ramp_time_fall  Defines the deceleration ramp time. This parameter should be used if the accel_flag is set to CX_CONST_RAMP_TIME. (READ ONLY)

accel_rise  Defines the acceleration rate. This parameter is used if the accel_flg is set to CX_CONST_RAMP_ACCEL.

accel_fall  Defines the deceleration rate. This parameter is used if the accel_flg is set to CX_CONST_RAMP_ACCEL.

screw_accel_rise  Defines the screw acceleration rate. This parameter is defined in rad/sec^2. (READ ONLY)

screw_accel_fall  Defines the screw deceleration rate. This parameter is defined in rad/sec^2. (READ ONLY)

rise_S_time  Defines the S-Curve acceleration time (see documentation on the CxSetSAccelTimes API in the “Motion” library in the Programmer’s Reference Manual). (READ ONLY)

fall_S_time  Defines the S-Curve deceleration time (see documentation on the CxSetSAccelTimes API in the “Motion” library in the Programmer’s Reference Manual). (READ ONLY)

interp_type  Defines the motion interpolation type to be used on the current move. Possible values include:
CX_JOINT_INTERP
CX_LINEAR_INTERP
CX_CIRCULAR_INTERP (READ ONLY)

mid_arc  A matrix defining a frame at an intermediate pose along a circular arc. (READ ONLY)

The developer can pass any field in this data structure to the required device interface functions to successfully control the mechanism.

The CxController and cmd_msg Data Structures
The open architecture of CODE allows both mechanism and I/O controllers to be integrated into the CIMServer software through the interface defined in this chapter. Many such controllers support
commands and functionality not supported directly by the CIMServer. The CxSendDeviceCommand API was implemented to allow you to send controller specific commands to a device controller, and receive appropriate responses. Each mechanism and group of I/O signals defined in the CIMServer’s database are associated with a controller when run in an on-line mode. The CxController data structure is used to define an instance of a specific controller, and is returned to a CODE application process when the CxGetControllerFromMech or the CxGetControllerFromSig API’s are called. The elements of this data structure are defined as follows:

```c
typedef struct cntrl_msg {
    long mtype ;
    long stype ;
    long size_in_bytes;
    struct CxServerId *which_server ;
    struct mechanism_entry *mech ;
    long sig_id ;
    long controller_index ;
    caddr_t driver_info; /* Generic pointer control specific info */
    long ret_code ; /* 0 ( Successful ) or -1 ( Error ) */
    long error ;  /* Error code */
} cntrl_msg ;
typedef cntrl_msg *CxController ;
```

The elements of this data structure are defined when either the CX_CNTRL_GET_MECHANISM_CNTRL or the CX_CNTRL_GET_SIGNAL_CNTRL functions are invoked. These functions are called when the CxGetControllerFromMech or the CxGetControllerFromSig API’s are called from a CODE application, respectively. This data structure is returned back to the corresponding CODE application process, and is later used with the CxSendDeviceCommand API.

In this data structure, only one field is important:

- `driver_info`. This field should be used to reference some information which uniquely defines access to the device controller. For example, the `mech->cntrl_info` field could be used for a mechanism controller, or the `signals_t->sdrvr->drvrInfo` field could be used for an I/O controller.

When the CxSendDeviceCommand API is called a cmd_msg data structure is sent to the server. This data structure is defined as follows:
typedef struct cmd_msg {
    long mtype ;
    long stype ;
    long size_in_bytes ;

    struct CxAddress *client_on_host ;

    long controller_index ; /* Index into CxController func
        list */
    caddr_t driver_info ; /* Generic pointer to controller
        specific info */
    long on_queue ; /* (T/F) Place on queue or proc.
        immediately ( currently not
        implemented ) */
    long cmd_len ; /* Length of command string */
    long ret_code ; /* 0 ( Successful ), or -1 ( Error
        ) */
    long error ; /* Error code */
    char cmd[MAX_CMD_STR] ; /* Command string */
} cmd_msg ;

The pertinent fields in this data structure are defined as follows:

**driver_info**  This field is used to reference some information which
    uniquely defines access to the device controller. It is the
    same information referenced in the
    CX_CNTRL_GET_MECHANISM_CNTRL or
    CX_CNTRL_GET_SIGNAL_CNTRL functions. (READ
    ONLY)

**cmd_len**  Defines the length (ie. number of bytes) reference in the
    cmd buffer. (READ WRITE)

**cmd**  Defines a character buffer containing controller specific
    information which will be sent to the device controller.
    (READ WRITE)

### The controller_instance Data Structure

The controller_instance data structure is used to identify the user defined functions developed to support
a particular I/O or motion controller. This data structure is defined as follows:

typedef struct controller_instance {
    long controller_type ;
long (*motion_parm_func)( mechanism_entry *mech, long parm_type, 
    parm_data *parm, CxErrorMsg *this_error ) ;

long (*motion_func)( mechanism_entry *mech, long function, void 
    *arg, CxErrorMsg *this_error ) ;

long (*signal_func)( signals_t *sigPtr, long function, void *arg, 
    CxErrorMsg *this_error ) ;

long (*signal_drvr_func)( signal_drivers_t *drvrPtr, long 
    function, void *arg, CxErrorMsg *this_error ) ;

long (*send_cmd)( cmd_msg *msg, CxErrorMsg *err ) ;

} controller_instance ;

These fields are described as follows:
controller_type  This field is set to the user defined controller type. This parameter is defined for a mechanism using the CxSetControllerType API in a CODE application process. The value placed in this field should be defined in the $ROBTOP/include/code/cntr_const.h file (where $ROBTOP defines the installation directory) (READ ONLY)

motion_parm_func  This field defines a function pointer to a user defined function used to set or get motion control parameters (i.e. xxxx_motion_parm). The possible parameters are defined by the parm parameter (see “Controller Interface Functions” on page 4-21) (READ ONLY)

motion_func  This field defines a function pointer to a user defined function used to control mechanism motions (i.e. xxxx_motion_func). The action taken by this function is defined by the function argument (see“Controller Interface Functions” on page 4-21 for possible values) (READ ONLY)

signal_func  This field defines a function pointer to a user defined function used to implement I/O signal specific functions (i.e. xxxx_signal_func). The action taken by this function is defined by the function argument (see “Controller Interface Functions” on page 4-21 for possible values). (READ ONLY)

signal_driver_func  This field defines a function pointer to a user defined function used to implement I/O driver interface functions (i.e. xxxx_signal_driver_func). The action taken by this function is defined by the function argument (see “Controller Interface Functions” on page 4-21). (READ ONLY)

send_cmd  This field defines a function pointer to a user defined function used to implement controller functions not supported by the CIMServer (i.e. xxxx_send_cmd).

Controller Interface Functions
This section contains a detailed description of each function’s required calling parameters, functionality, and required return codes to interface correctly with the CIMServer.

Adding Motion and I/O Controller Interface Functions
User defined motion and I/O control functions are defined in the file:
There are five functions required to interface with a mechanism or I/O controller. These were previously described in “Developing Controller Interfaces” on page 4-4. They include:

- \texttt{xxxx\_motion\_parm}
- \texttt{xxxx\_motion\_func}
- \texttt{xxxx\_signal\_func}
- \texttt{xxxx\_signal\_drv\_func}
- \texttt{xxxx\_send\_cmd}

**NOTE:** You must replace \texttt{xxxx} with your own prefix. Prefixes \texttt{pmac, soft, OAC, pcl722, ac28,}
and \texttt{ibs} are already in use. You should check the \texttt{user\_cntrl.c} module to ensure your prefix does not conflict with existing entries. Duplication of a prefix will prevent your custom CIMServer from compiling.

To integrate your implementation of these functions, execute the following steps:

1. Examine the syntax for each function, as described in the following sections. Implement the functionality supported by your motion or I/O control interface.

2. Assign a constant for your specific controller implementation. Cimetrix reserves numbers between 0 and 999 for Cimetrix developed drivers. Therefore, the constant you choose should be greater than 999. Add a definition of this constant in the file $\texttt{ROBTOP/include/code/cntr\_const.h}$

   Example:
   ```
   /* Driver constants 0 through 999 are reserved for Cimetrix use */
   #define CX\_SOFTWARE 0
   #define CX\_PMAC\_DRVR 1
   #define CX\_MEI\_DRVR 2
   #define CX\_ANORAD\_VPC\_2000 3
   #define CX\_IBS\_DRVR 4
   #define CX\_PCL722\_DRVR 5
   #define CX\_AC28\_DRVR 6
   #define CX\_PDX\_DRVR 7
   #define CX\_CIM\_OAC 8
   
   /* User Driver constants 1000 on */
   #define MY\_DRIVER 1000
   ```

3. Edit the file $\texttt{ROBTOP/lib/cimetrix/custom/user\_cntrl.c}$.
   a. Add any necessary header files with function prototypes for your specific functions.
   b. Increment the constant \texttt{CX\_NUM\_CONTROLLER} by a value of 1.
   c. Add your functions to the array of \texttt{controller\_instance} data structures (see “The controller\_instance Data Structure” on page 4-19 for a description of this data structure). If you
do not support one of the defined functions in your driver, set the corresponding function pointer to a CX_NULL pointer.

Example:

```c
#define CX_NUM_CONTROLLER 4
controller_instance controller_funclist[CX_NUM_CONTROLLER] = {
    {
        CX_SOFTWARE,
        soft_motion_parm,
        soft_motion_func
        soft_signal_func,
        soft_signal_drvr_func,
        soft_send_cmd
    },
    {
        CX_PMAC_DRVR,
        #ifdef USE_PMAC
        pmac_motion_parm,
        pmac_motion_func,
        pmac_signal_func,
        pmac_signal_drvr_func,
        pmac_send_cmd
        #else
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
        #endif
    },
    {
        CX_CIM_OAC,
        #ifdef USE_CIM_OAC
        OAC__motion_parm,
        OAC_motion_func,
        OAC_signal_func,
        OAC_signal_drvr_func,
        NULL
        #else
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
        #endif
    },
```
{ MY_DRIVER, my_drvr_motion_parm, my_drvr_motion_func, my_drvr_signal_func, my_drvr_signal_drvr_func, my_drvr_send_cmd }

4. Modify the makefile in the $ROBTOP/lib/cimetrix/custom directory to incorporate libraries and object files required to compile and link your custom functions.

5. Type gmake to recompile a version of the CIMServer with your extension included.

**Implementing the xxxx_motion_parm Function**

The xxxx_motion_parm function has the following syntax:

```c
long xxxx_motion_parm ( mechanism_entry *mech,
                        long parm_type,
                        parm_data *parm,
                        CxErrorMsg *this_error )
```

The arguments are defined as follows:

- `mechanism_entry *mech` A pointer to a mechanism_entry data structure (see “The mechanism_entry Data Structure” on page 4-10).
- `long parm_type` A constant defining the parameter type. The possible parameter types are described below.
- `parm_data *parm` A pointer to a parm_data data structure. This data structure is used to pass motion parameters between the interface functions and the CIMServer motion algorithms, and is defined as follows:

```c
typedef parm_data {
    union {
        double dval ;
        double *dptr ;  /* Pointer to double value */
        long lval ;   /* Long value */
        long *lptr ;   /* Pointer to double value */
        unsigned long uval ; /* Unsigned long value */
    }
}
```

4-24
unsigned long *uptr ; /* Pointer to unsigned long */
void *vptr ; /* Void pointer */
} parm_union ;
} parm_data ;

CxErrorMsg *this_error
A pointer to a CxErrorMsg data structure.

This function should return 0 (CX_OK) upon successful completion, or -1 (CX_ERROR) if an error occurs. If an error occurs, the error should be handled according to instructions given in “Handling Driver Errors” on page 4-39.

The possible values for the parm_type parameter and their corresponding implementations are defined as follows:

**CX_CNTRL_GET_CURRENT_POSITION**

The CX_CNTRL_GET_CURRENT_POSITION function is used to obtain the current position of the mechanisms actuators. The number of mechanism axes is defined in mech->rob->ndof. The current actuator positions are stored in an array of double values. The pointer to this array is accessed through the parm_data data structure as parm_data->parm_union.dptr.

Once this data is obtained, the CIMServer transforms the actuator positions into joint positions, using the actuator_to_joint conversion function defined for the mechanism.

This function is invoked under the following conditions:

1. When the CxGetAxes() API function call is made.
2. When the mechanism is opened for either CX_MONITOR, or CX_CONTROL, for the first time.
3. When the CxUpdateJoints() API function is called.
4. At the completion of the CxHomeMechanism() API function.
5. After the servo amplifiers have been successfully enabled using the CxEnableAmps() API function, and the mechanism has been successfully homed.
6. When a halted motion is aborted using the CxSendMechanismErrorAction() API function.

**CX_CNTRL_GET_LATCHED_POSITION**

The CX_CNTRL_GET_LATCHED_POSITION function is used to obtain the latched positions of the mechanisms actuators. The number of mechanism axes is defined in mech->rob->ndof. The current actuator positions are stored in an array of double values. The pointer to this array is accessed through the parm_data data structure as parm_data->parm_union.dptr.
Once this data is obtained, the CIMServer transforms the actuator positions into joint positions, using the actuator_to_joint conversion function defined for the mechanism. This function is invoked under the following conditions:

1. When a call is made to the `CxGetLatchedAxes()` API function.

**CX_CNTRL_GET_MOTION_STATUS**

The **CX_CNTRL_GET_MOTION_STATUS** function is used to obtain the current motion status of the mechanism. The motion status can be set to one of the following values:

- **CX_IN_MOTION** The mechanism is in motion.
- **CX_NOT_IN_MOTION** The mechanism is not in motion.

The current motion status is stored in a long parameter, referenced in the `parm_data` data structure as `parm_data->parm_union.lval`. This function is called under the following conditions:

1. When the `CxGetMotionStatus()` API function is called in a CODE application.

**CX_CNTRL_GET_CONTROLLER_STATUS**

The **CX_CNTRL_GET_CONTROLLER_STATUS** function is used to determine the current status of the controller. The possible controller states are defined as follows:

- **CX_CONTROLLER_INITIALIZED** Has the controller been initialized?
- **CX_AMPS_ENABLED** Are the servo amplifiers enabled?
- **CX_MECHANISM_HOMED** Has the mechanism been homed?
- **CX_TEACH_PENDANT_ACTIVE** Is the controller’s native teach pendant active?
- **CX_KEY_SWITCH_ACTIVE** Is a key switch associated with the controller active?
- **CX_MECHANISM_HALTED** Is the mechanism in a halted state?

The desired status parameters are passed into the function as a bitmask contained in the `parm_data` data structure as `parm_data->parm_union.lval`. Once the requested states have been checked, if the condition remains **CX_TRUE**, the mask should remain set. If the requested state is **CX_FALSE**, the mask should be cleared. This function is called under the following conditions:

1. When the `CxGetControllerStatus()` API function is called in a CODE application.
2. When the `CxEnableAmps()` API function is called in a CODE application.
3. When the `CxUpdateJoints()` API function is called in a CODE application.
4. When an I/O signal is triggered, and a CODE application had registered an interrupt on that I/O signal (i.e. using the `CxInterruptOnChange()`, `CxInterruptOnValue()`, `CxInterruptOnUpperThreshold()`, or `CxInterruptOnLowerThreshold()`), and the mechanism is associated with this interrupt, the function will be called to determine if the mechanism is a halted state.

5. When a mechanism is opened for `CX_CONTROL` and the `CX_KEY_PENDANT` or `CX_KEY_NORMAL` masks are defined.

6. When the `CxStopMechanism()` API function is called in a CODE application, to determine if the mechanism is already halted.

7. When the `CxSendMechanismErrorAction()` API function is called to abort or resume a motion, and the trajectory for the current motion is being calculated by the CIMServer, this function is used to determine if the mechanism is already halted.

**CX_CNTRL_GET_CONTROLLER_ATTRIBUTES**

The `CX_CNTRL_GET_CONTROLLER_ATTRIBUTES` function is used to determine the controller’s capabilities. The possible controller attributes are defined as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_TRAJECTORY_MODE</td>
<td>The motion control interface supports low level trajectory data from the CIMServer.</td>
</tr>
<tr>
<td>CX_CNTRL_TELE_MODE</td>
<td>The motion control interface supports its own tele-operated control interface.</td>
</tr>
<tr>
<td>CX_INVKIN_MODE</td>
<td>The motion control interface can generate Cartesian trajectories, such as <code>CX_LINEAR_INTERP</code> or <code>CX_CIRCULAR_INTERP</code>.</td>
</tr>
<tr>
<td>CX_FWDSDKIN_TIME_MODE</td>
<td>The motion control interface supports time based <code>CX_JOINT_INTERP</code> trajectory generation.</td>
</tr>
<tr>
<td>CX_FWDSDKIN_RATE_MODE</td>
<td>The motion control interface supports rate based <code>CX_JOINT_INTERP</code> trajectory generation.</td>
</tr>
<tr>
<td>CX_TEACH_PENDANT_MODE</td>
<td>The motion control interface supports its own native teach pendant.</td>
</tr>
<tr>
<td>CX_FWDSDKIN_BLEND_MODE</td>
<td>The motion control interface supports blending <code>CX_JOINT_INTERP</code> type moves while doing native trajectory generation.</td>
</tr>
<tr>
<td>CX_INVKIN_BLEND_MODE</td>
<td>The motion control interface supports blending while performing its own Cartesian trajectory generation.</td>
</tr>
</tbody>
</table>

These parameters are bitmasks which should be set directly in the `mech->controller_attr` field.
This function is called under the following conditions:

1. When the CxGetControllerAttr() API function is called in a CODE application.
2. When a mechanism is opened using the CxOpenMechanism() API function in a CODE application.

**CX_CNTRL_GET_ERROR_STATUS**

The **CX_CNTRL_GET_ERROR_STATUS** function is used to determine an asynchronous error condition on the motion control interface. This function can operate in a blocking or non-blocking mode. The function should block for an error condition if the `parm_data->parm_union.lval` parameter is set to **CX_TRUE**. Otherwise the function should return the error status immediately.

The error condition should be logged in the `CxErrorMsg *this_error` data structure (see “Handling Driver Errors” on page 4-39 for details on handling driver errors).

This function is called under the following conditions:

1. If the threaded version of the CIMServer is being used, and the mechanism is opened for **CX_CONTROL** using the CxOpenMechanism() API function, this function will be called. If an asynchronous error occurs, this function should wake up and return. As soon as the CIMServer has handled the error, the function will be called again.
2. If the non-threaded version of the CIMServer is being used, and the mechanism is opened for **CX_CONTROL** using the CxOpenMechanism() API function, this function will be called each event cycle to determine if an asynchronous error has occurred.

**CX_CNTRL_GET_MECHANISM_CNTRL**

The **CX_CNTRL_GET_MECHANISM_CNTRL** function is used to obtain generic information about the mechanism controller for use with the CxSendDeviceCmd() API function. Information allowing the CxSendDeviceCmd() function to communicate with the motion controller is stored in the `CxController` data structure in the `driver_info` field (see “The CxController and cmd_msg Data Structures” on page 4-17). A pointer to the `CxController` data structure is passed in as `parm_data->parm_union.vptr`. The content of this data structure can then be de-referenced as necessary.

This function is called under the following conditions:

1. When a call is made to the CxGetControllerFromMech() API function call in a CODE application.

**Implementing the xxxx_motion_func Function**

The `xxxx_motion_func` function has the following syntax:

```c
long xxxx_motion_func ( mechanism_entry *mech,
                        long function,
                        void *arg,
                        CxErrorMsg *this_error ) ;
```

The arguments are defined as follows:
mechanism_entry *mech  A pointer to a mechanism_entry data structure (see “The mechanism_entry Data Structure” on page 4-10).

long function  A constant describing the function to be performed. Possible values are described below.

void *arg  A void pointer which is interpreted depending on the value of the function parameter.

CxErrorMsg *this_error  A pointer to a CxErrorMsg data structure.

This function should return 0 (CX_OK) upon successful completion, or -1 (CX_ERROR) if an error occurs. If an error occurs, the error should be handled according to instructions given in “Handling Driver Errors” on page 4-39.

The possible values for the function parameter, and their corresponding implementation are described as follows:

**CX_CNTRL_STOP_MOTION**

The CX_CNTRL_STOP_MOTION function is used to stop a mechanism if it is in motion. The arg parameter is mapped to a long value which can be either CX_TRUE, or CX_FALSE. If arg is CX_TRUE, the function should block, and wait for motion to be completely stopped before returning. If arg is CX_FALSE, halting motion should be initiated, but the function should return immediately without waiting for motion to come to a complete stop.

Once stopped, the user must call the CxSendMechanismErrorAction API with the appropriate error action defined, in order to abort or restart the motion. The CX_CNTRL_RESUME_MOTION function is used to restart a stopped motion. The CX_CNTRL_FLUSH_MOTION_QUEUE function is used to abort the current motion and flush all pending motion in the controller’s motion queue. This function is called under the following conditions:

1. When the CxStopMechanism API function is called from a CODE application process, and the trajectory mode of the current motion is set to CX_END_POINT_MOVE.

2. When an I/O signal is triggered which is being monitored in a CODE application process (i.e. using CxInterruptOnChange, CxInterruptOnValue, CxInterruptOnLowerThreshold, or CxInterruptOnUpperThreshold); the current trajectory mode is defined to be an CX_END_POINT_MOVE; and the mechanism is associated with the interrupt condition.

**CX_CNTRL_WAIT_FOR_MOTION_STOP**

The CX_CNTRL_WAIT_FOR_MOTION_STOP function is used to wait for any current motion to complete. The arg parameter is passed in as a long value, and can have the values CX_TRUE, or CX_FALSE. If arg is CX_TRUE, then this function should wait until the current motion is complete. If arg is CX_FALSE, the function should return CX_TRUE if all queued motions have completed, and the mechanism is stopped. Otherwise, it should return CX_FALSE if the mechanism is still in motion.
This function is called under the following conditions:

1. When the \texttt{CxSetServerType()} function is called to transition from \texttt{CX\_RUNTIME} to \texttt{CX\_ANIMATION}, this function is used to ensure that all on-line motions are complete before transitioning to the \texttt{CX\_ANIMATION} mode.

2. When the \texttt{CxWaitForEndOfMotion()} API function is called, this function is used to ensure that all queued moves on the card are complete.

3. If a \texttt{start\_signal} is set using the \texttt{CxSetStartSignal()} API function prior to initiating a motion, this function is used to ensure that all prior motions are complete before setting the \texttt{start\_signal}.

4. If the \texttt{blend\_policy} is set to \texttt{CX\_MOVE\_TO} or \texttt{CX\_MOVE\_WAIT}, and the \texttt{trajectory\_mode} is set to \texttt{CX\_CALCULATE\_TRAJECTORY}, this function is used to ensure that all queued motions have completed before starting the next motion.

5. If the \texttt{blend\_policy} is set to \texttt{CX\_MOVE\_WAIT}, and the \texttt{trajectory\_mode} is set to \texttt{CX\_END\_POINT\_MOVE}, this function is used to wait for motion to complete before responding to the client application.

6. When a transition is made between a \texttt{CX\_CALCULATE\_TRAJECTORY} move and a \texttt{CX\_END\_POINT\_MOVE} motion.

7. If the \texttt{stop\_signal} is set using the \texttt{CxSetStopSignal()} function, and the \texttt{blend\_policy} for the current move is set to \texttt{CX\_MOVE\_TO} or \texttt{CX\_MOVE\_WAIT}, the motion must be completely finished before setting the \texttt{stop\_signal}.

8. When the \texttt{CxUpdateJoints()} API function is called, this function is used to ensure that motion has stopped before obtaining the current mechanism position.

9. When the \texttt{CxHomeMechanism()} API function is called, this function is used to ensure that all queued motion has stopped before initiating the homing sequence. It is also used to wait for the homing sequence to be completed before responding back to the CODE application.

10. When the \texttt{CxJogRobot()} API function is called, this function is used to ensure that all queued motions on the controller have completed before allowing the native teach pendant to control the mechanism. It can also be used to wait for the native teach pendant to relinquish control of the mechanism.

11. When the \texttt{CxCloseMechanism()} API function is called, this function is used to ensure that all queued motions have completed before severing the connection to the controller.

12. When the \texttt{CxSendMechanismErrorAction()} API function is called, this function is used to ensure that queued motion are completed before the current location of the mechanism is updated.
**CX_CNTRL_CLOSE_CONTROLLER**
The `CX_CNTRL_CLOSE_CONTROLLER` function severs the connection to the mechanism controller. The `arg` parameter is not used with this function, and is passed in as a `CX_NULL` pointer.
This function is called under the following conditions:

1. When the `CxCloseMechanism()` API function is called from a CODE application
2. When the Server dies unexpectedly.

**CX_CNTRL_AMPLIFIER_CONTROL**
The `CX_CNTRL_AMPLIFIER_CONTROL` function is used to enable or disable the mechanism’s servo amplifiers. The `arg` parameter is passed in as a long value. When set to `CX_TRUE`, the function should enable the servo amplifiers. When set to `CX_FALSE` the mechanism’s servo amplifiers should be disabled.
This function is called under the following conditions:

1. When the `CxEnableAmps()` function is called to enable the servo amplifiers.
2. When the `CxDisableAmps()` function is called to disable the servo amplifiers.

**CX_CNTRL_INITIATE_HOME_SEQUENCE**
The `CX_CNTRL_INITIATE_HOME_SEQUENCE` is used to initiate the homing sequence for the specified mechanisms. The `arg` parameter is not used with this function, and is passed in as a `CX_NULL` pointer.
This function is called under the following conditions:

1. When the user calls the `CxHomeMechanism` API from a CODE application.

**CX_CNTRL_INVOKE_TEACH_PENDANT**
The `CX_CNTRL_INVOKE_TEACH_PENDANT` is used to invoke the native teach pendant associated with a mechanism controller. When called, control of the mechanism should be turned completely over to the controller’s teach pendant, and should only return once the controller has relinquished control of the mechanism.
This functions is called under the following conditions:

1. When the `CxJogRobot()` API function is called from a CODE application.

**CX_CNTRL_OPEN_CONTROLLER**
The `CX_CNTRL_OPEN_CONTROLLER` functions establishes a connection with the mechanism controller. The `arg` parameter is passed in as a long value and defines whether the mechanism should be opened for `CX_CONTROL`, or `CX_MONITOR`.
If the internal description of the mechanism controller requires a special data structure, memory for that data structure should be allocated in this function, and the pointer to that data structure can be assigned to the `cntrl_info` field in the `mechanism_entry` data structure.
This function is called under the following conditions:
1. When the CxOpenMechanism() API function is called from a CODE application.

**CX_CNTRL_FLUSH_MOTION_QUEUE**
The CX_CNTRL_FLUSH_MOTION_QUEUE function is used to abort a halted motion, and flush all pending motion requests in the controller’s motion queue. All error conditions should also be rectified so that the controller is prepared to accept new motion requests. The arg parameter is not used with this function, and is passed in as a CX_NULL parameter.
This function is called under the following conditions:

1. When the CxSendMechanismErrorAction() API function is called with CX_MECH_ABORT as the argument.

**CX_CNTRL_RESUME_MOTION**
The CX_CNTRL_RESUME_MOTION function is used to restart a halted motion, which was stopped with the CX_CNTRL_STOP_MOTION function. The arg parameter is not used with this function, and is passed in as a NULL parameter.
This function is invoked under the following conditions:

1. When a CX_END_POINT_MOVE type move is being resumed using the CxSendMechanismErrorAction() API function is called from a CODE application.

**CX_CNTRL_SEND_MOVE**
The CX_CNTRL_SEND_MOVE function is used to send move commands to the controller. A pointer to a command_entry data structure is passed into the function through the arg parameter. The command_entry data structure contains all of the information required to complete the move, such as the speeds, accelerations, interpolation types, etc. (see “The command_entry Data Structure” on page 4-12 for detailed data structure information).
The command_entry->trajectory_mode field will be set to either CX_CALCULATE_TRAJECTORY or CX_END_POINT_MOVE.
If the command_entry->trajectory_mode is set to CX_END_POINT_MOVE, the trajectory generation is done by the servo controller, and the controller needs to support the requested interpolation type, blend policy, tool motion type, etc.; otherwise, an error condition should be returned.
If the command_entry->trajectory_mode field is set to CX_CALCULATE_TRAJECTORY, then trajectory generation is performed by the CIMServer and position, velocity, and/or time (i.e. PVT) information can be sent to the servo controller at the trajectory rate defined in the CIMServer. In this case the command_entry->time_step parameter is the time required to move to the commanded target which is calculated by the CIMServer’s trajectory generator.
The target joint values are defined in the command_entry->dof_act parameter, which is a pointer to an array of double values. The desired speed for each actuator is defined in the command_entry->curspd_act field, and is also a pointer to an array of double values.
This function should return a value as soon as the move has been accepted into the servo controller’s motion queue. If the blend policy is set to CX_MOVE_WAIT, then CX_CNTRL_WAIT_FOR_MOTION_STOP function is used to determine if a motion has been completed.
This function is called under the following conditions:
1. When any CODE API function is called which initiates motion (e.g. CxMoveSingleAxis, CxMoveToNode, CxMoveToConfig, etc.).

**CX_CNTRL_INITIALIZE_CONTROLLER**

The CX_CNTRL_INITIALIZE_CONTROLLER function is used to initialize any controller specific parameters or defaults before any interaction with the controller. The arg parameter is not used with this function, and is passed in as a NULL pointer. This function is called under the following conditions:

1. After a controller has been opened (using CxOpenMechanism API function), but before any motion commands are sent to the controller. It is used to initialize any controller.

**Implementing the xxxx_signal_func Function**

The xxxx_signal_func function has the following syntax:

```
long xxxx_signal_func ( signals_t *sigPtr, long function, void *arg, CxErrorMsg *this_error );
```

The arguments are defined as follows:

- `signals_t *sigPtr` A pointer to a signals_t data structure, which contains information about the specified I/O signal (see “The signals_t Data Structure” on page 4-6 for detailed explanation of the elements in this data structure).
- `long function` The action to be performed by the xxxx_signal_func function.
- `void *arg` A void pointer whose value depends on the function parameter.
- `CxErrorMsg *this_error` A pointer to a CxErrorMsg data structure, used for reporting errors to the calling function.

This function should return 0 (CX_OK) upon successful completion, or -1 (CX_ERROR) if an error occurs. If an error occurs, the error should be handled according to instructions given in “Handling Driver Errors” on page 4-39.

The possible values for the function parameter are defined as follows:

**CX_CNTRL_INITIALIZE_SIGNAL**

The CX_CNTRL_INITIALIZE_SIGNAL function is used to initialize the state of a hardware signal. The signal is defined by the sigPtr parameter.

This function is called under the following conditions:

1. When the CIMServer reads the signal table and initializes each signal.
**CX_CNTRL_GET_SIGNAL_VALUE**

The **CX_CNTRL_GET_SIGNAL_VALUE** function is used to get the value of a hardware signal. The `arg` parameter is passed into the `xxxx_signal_func` function as a pointer to a long value, and the current signal value should be stored here.

This function is called under the following conditions:

1. When the CIMServer loads the signal table and initializes signals, any signals with the `input` flag set (i.e. to 1-polled, or 2-interrupt) need to be read to determine their current states.

2. When signal monitoring is enabled (using the `CxInterruptOnChange`, `CxInterruptOnValue`, `CxInterruptOnUpperThreshold`, `CxInterruptOnLowerThreshold`, `CxWhen`, `CxWaitForValue`, `CxWaitForChange`, `CxWaitForUpperThreshold`, or `CxWaitForLowerThreshold` CODE API functions), and the specified signal requires polling, this function is used to poll the signal’s current value.

3. When the `CxGetSignalValue()` CODE API function is called.

**CX_CNTRL_SET_LATCH**

The **CX_CNTRL_SET_LATCH** function is used when a mechanism controller can determine its current joint values when a hardware latchable signal changes its state.

The `arg` parameter contains a pointer to a `signal_info` data structure (see “The signal_info Data Structure” on page 4-9).

In the `signal_info` structure, `signal_info->mech` is a pointer to a mechanism whose joint values are to be determined when the signal changes its state.

The `signal_info->change` parameter defines the I/O conditions under which the mechanism’s position should be latched. Its possible values are defined as follows:

- **LATCH_RISE_FLAG**
  - Latch on the rising edge of the specified signal.

- **LATCH_FALL_FLAG**
  - Latch on the falling edge of the specified signal.

- **LATCH_RISE_FLAG_RISE_CHNC**
  - Latch on the rising edge of the specified signal and the rising edge of the encoder’s index pulse (or third channel).

- **LATCH_RISE_FLAG_FALL_CHNC**
  - Latch on the rising edge of the specified signal and the falling edge of the encoder’s index pulse (or third channel).

- **LATCH_FALL_FLAG_RISE_CHNC**
  - Latch on the falling edge of the specified signal and the rising edge of the encoder’s index pulse (or third channel).

- **LATCH_FALL_FLAG_FALL_CHNC**
  - Latch on the falling edge of the specified signal and the falling edge of the encoder’s index pulse (or third channel).
LATCH_RISE_CHNC  Latch on the rising edge of the encoder's index pulse (or third channel).
LATCH_FALL_CHNC  Latch on the falling edge of the encoder's index pulse (or third channel).
LATCH_SOFTWARE  Latch on a non-hardware latchable signal. In this case, the latch may not be as efficient as the other types described above.

This function is invoked under the following conditions:
1. When the CxLatchOnTrigger API is called from a CODE application.

**CX_CNTRL_SET_SIGNAL_VALUE**
The CX_CNTRL_SET_SIGNAL_VALUE function is used to set the state of a hardware signal. The desired signal value is passed in through the arg parameter as a long value.
This function is called under the following conditions:
1. When the CxSetSignal() API function is called from a CODE application.
2. When the signal table is loaded, a signal is defined as an output, and the use_init field is set for that signal.

**CX_CNTRL_SETUP_SIGNAL_MONITORING**
The CX_CNTRL_SETUP_SIGNAL_MONITORING function is used to define how the controller will monitor the state of a given signal and what action to take when the state of the signal changes. The actual monitoring of an I/O sub-system is set up using the CX_CNTRL_MONITOR_SIGNALS function (see “CX_CNTRL_MONITOR_SIGNALS” on page 4-37).
This function is used only by the threaded version of the CIMServer, since the function CX_CNTRL_MONITOR_SIGNALS blocks until a desired state change occurs. In the Non-Threaded version, each signal must be polled to determine if its state has changed.
In addition, this function can be used only for I/O drivers which can generate interrupts when a state change occurs. All other types of signals must be polled, since their drivers cannot generate asynchronous interrupts.
The arg parameter is passed in as a pointer to a signal_info data structure (see “The signal_info Data Structure” on page 4-9). The signal_info->comparison parameter defines how to monitor a signal or group of signals. Its possible values are the following:

```c
  CX_DONT_MONITOR  Disables signal monitor.
  CX_ANY_VALUE_CHANGE  Causes operation to take place if the signal changes state.
  CX_SPECIFIC_VALUE_CHANGE  Causes operation to take place if the signal reached a defined state specified by the value parameter.
```
**CX_GREATER_THAN_CHANGE**
Causes operation to take place if the signal becomes greater than the value specified by the value parameter.

**CX_LESS_THAN_CHANGE**
Causes operation to take place if the signal becomes less than the value specified by the value parameter.

**CX_ANY_SIGNAL_CHANGE**
Causes operation to take place if any signal changes state. If this is defined, then definition of the specific signal should be ignored.

The `signal_info->operation` parameter defines what action to take when the condition specified in the change field occurs. Its possible values are the following:

**CX_INTERRUPT_ON_TRIGGER**
Notify the driver when the specified signal reaches its state.

**CX_HALT_ON_TRIGGER**
Halt the mechanism specified by the mech parameter when the signal is triggered, and notify the driver that the signal has reached the state defined by the comparison parameter.

The `signal_info->mech` parameter defines a mechanism associated with the signal event being monitored, which when triggered, will cause the defined motion to stop. If this parameter is `CX_NULL`, no mechanism should be halted when the event is triggered.

This function is called under the following conditions:

1. **When event monitoring is enabled using** the `CxInterruptOnChange`,  
   `CxInterruptOnValue`, `CxInterruptOnUpperThreshold`,  
   `CxInterruptOnLowerThreshold`, `CxWhen`, `CxWaitForValue`, `CxWaitForChange`,  
   `CxWaitForUpperThreshold`, or `CxWaitForLowerThreshold` **CODE API functions**.

2. **When event monitoring is disabled for a specified signal using** the  
   `CxInterruptOnChangeOff`, `CxInterruptOnValueOff`,  
   `CxInterruptOnUpperThresholdOff`, or `CxDisableWhen` **CODE API functions**.

**CX_CNTRL_GET_SIGNAL_CNTRL**
The `CX_CNTRL_GET_SIGNAL_CNTRL` function is used to get controller specific information, which is later used by the `xxxx_send_cmd` function. The `arg` parameter contains a pointer to a `CxController` data structure (see “The `CxController` and `cmd_msg` Data Structures” on page 4-17). The `CxController->driver_info` field should be set in the `CxController` data structure before returning. It can be defined as a pointer to a controller specific data structure allowing the `xxxx_send_cmd` function to communicate with the host controller.

This function is called under the following conditions:
1. When the CxGetControllerFromSignal() API function is called from a CODE application.

Implementing the xxxx_signal_drvr_func Function

The xxxx_signal_drvr_func function has the following syntax:

```
long xxxx_signal_drvr_func( signal_drivers_t *drvrPtr, long function, void *arg, CxErrorMsg *this_error ) ;
```

The arguments are defined as follows:

- `signal_drivers_t *drvrPtr`: A pointer to a signal_drivers_t data structure (see “The signal_drivers_t Data Structure” on page 4-8).
- `long function`: A long parameter describing the function to be performed. Possible values are described below.
- `void *arg`: A void pointer containing information required to perform the corresponding function. This parameter is interpreted depending on the function to be performed.
- `CxErrorMsg *this_error`: A pointer to a CxErrorMsg data structure.

This function should return 0 (CX_OK) upon successful completion, or -1 (CX_ERROR) if an error occurs. If an error occurs, the error should be handled according to instructions given in “Handling Driver Errors” on page 4-39.

The possible values for the `function` parameter are defined as follows:

**CX_CNTRL_CLOSE_SIGNAL_TABLE**

The CX_CNTRL_CLOSE_SIGNAL_TABLE function is used to close a connection to a signal table on an I/O or mechanism controller. This function should not sever a connection with a mechanism controller if the I/O signal is associated with a mechanism controller.

This function is called under the following conditions:

1. When the CIMServer exits.

**CX_CNTRL_MONITOR_SIGNALS**

The CX_CNTRL_MONITOR_SIGNALS function is used by the threaded version of the CIMServer for handling I/O drivers which can generate asynchronous interrupts when a signal changes its state. In the non-threaded version of the CIMServer all input signals are continuously polled using the CX_CNTRL_GET_SIGNAL_VALUE function.

The manner in which a signal’s state is monitored is defined by the CX_CNTRL_SETUP_SIGNAL_MONITORING function.
The CX_CNTRL_MONITOR_SIGNALS function should block until a state change occurs. A pointer to a signal_info data structure (see “The signal_info Data Structure” on page 4-9) is passed into the function through the arg pointer. When the function returns, the signal_info->sigPtr parameter should be set to the signals_t pointer for the signal that was triggered. The signal_info->value parameter should contain the new value for that signal.

This function is called under the following conditions:

1. When the CIMServer is started, after the signal table has been loaded, and the signal monitor thread has been started for the I/O signal driver. When this function returns, the calling function will handle the I/O event by sending an appropriate response to a client process, and then call the CX_CNTRL_MONITOR_SIGNALS function again. This will continue until the CIMServer exits.

CX_CNTRL_OPEN_SIGNAL_TABLE

The CX_CNTRL_OPEN_SIGNAL_TABLE function is used to initialize the signal table on an I/O controller.

This function should make any calls required to initialize the I/O controller, and should then set the initialized field in the signal_drivers_t data structure to CX_TRUE. This can be referenced through the sdrvr pointer as drvrPtr->initialized.

This function should also insure that it does not re-initialize the hardware signal table if it has already been initialized by another Server, since several CIMServer can communicate with the same hardware I/O interface simultaneously.

This function is called under the following conditions:

1. When the first signal related to this I/O interface is initialized in the CIMServer signal table.

Implementing xxxx_send_cmd Function

The xxxx_send_cmd function has the following syntax:

```c
long xxxx_send_cmd ( cmd_msg *msg, CxErrorMsg *err ) ;
```

The arguments are defined as follows:

- `cmd_msg *msg` A pointer to a cmd_msg data structure (see “The CxController and cmd_msg Data Structures” on page 4-17 for detailed definition). The cmd_msg data structure is defined in $ROBTOP/include/code/msg_defs.h.

- `CxErrorMsg *err` A pointer to a CxErrorMsg data structure (see “Handling Driver Errors” on page 4-39).

The xxxx_send_cmd is used to send a controller specific command, which is not supported by the CIMServer, to the mechanism or I/O controller. This function is invoked when the CxSendDeviceCmd API is called from a CODE application process, in which the CxController structure was obtained using the CxGetControllerFromMech or CxGetControllerFromSig CODE API functions.

The msg->cmd parameter is defined as a generic pointer to a character string, and contains the controller specific command to be executed by the controller. Once the function has been executed, the
msg->cmd character buffer should be used to return the command execution result back to the calling process. The msg->cmd_len parameter contains the number of valid data bytes in the msg->cmd field. Once the function has been executed, this parameter should be set prior to returning. The msg->driver_info field contains information required to communicate with the controller. The information contained in this field was obtained in the CX_CNTRL_GET_SIGNAL_CNTRL and CX_CNTRL_GET_MECHANISM_CNTRL functions (see “CX_CNTRL_GET_SIGNAL_CNTRL” on page 4-36 and “CX_CNTRL_GET_MECHANISM_CNTRL” on page 4-28 respectively). This function should return 0 (CX_OK) upon successful completion, or -1 (CX_ERROR) if an error occurs. If an error occurs, the error should be handled according to instructions given in the next section, “Handling Driver Errors”.

### Handling Driver Errors

In order for your driver to function robustly, you must report errors back to the calling function. The CIMServer has an embedded set of utilities which allow you to add error handling to your custom driver interface.

If an error occurs in one of the functions described in previous sections, a CxErrorMsg data structure pointer is provided to record the error. The information in this structure is passed back to the application process, so that the error can be recorded and handled in an appropriate manner.

The CxErrorMsg data structure is defined in the file: $ROBTOP/include/code/error.h

The CIMServer provides a set of utility macros used for setting the parameters defined in this data structure. These are defined as follows:

#### SetErrorMsg

SetErrorMsg ( CxErrorMsg *this_error, long error_const ) ;
This macro initializes the contents of the CxErrorMsg data structure to zero (0) values (and should therefore be called first after an error occurs), and sets the error constant in the data structure to the value passed in as error_const. It also records the name of the source file in which the error occurred, the line number in the source file, and the time at which the error occurred.

The CIMServer reserves error constants in the range 0 to CX_BASE_USER_ERROR, which is defined in $ROBTOP/include/code/cntr_const.h. The CODE error constants are defined in the following header files:

%ROBTOP%/include\drivers\ac28\ac28_errno.h
%ROBTOP%/include\drivers\devnet\devnet_errno.h
%ROBTOP%/include\drivers\himc\himc_errno.h
%ROBTOP%/include\drivers\ipcnt\errors.h
%ROBTOP%/include\drivers\meids\mei_errors.h
%ROBTOP%/include\drivers\pcl722\pcl722_errno.h
%ROBTOP%/include\drivers\xmp\xmp_errors.h

Driver specific error constants should be defined in a separate header file, and should be defined below 0 or above CX_BASE_USER_ERROR.
SetErrorText

SetErrorText ( CxErrorMsg *this_error, char *text ) ;
This macro sets the error text for the error which has occurred. This macro should be called after the
SetErrorMsg macro. This text should be descriptive of the error, and can be up to
CX_ERROR_MSG_SZ bytes in length.

SetErrorMech

setErrorMech ( CxErrorMsg *this_error,
struct mechanism_id_msg *client_mech,
long jnt,
long action) ;

This macro is used to define mechanism specific errors.
The client_mech parameter refers to the CxMechanism pointer defined on the client, and is
referenced in the client_mech field in the mechanism_entry data structure (i.e. mech->client_mech).
The jnt parameter is used if the error occurred on a specific mechanism joint (e.g. a specific joint at a
limit). If the error does not apply to a specific joint, this parameter should be set to -1.
The action parameter defines whether or not (i.e. CX_TRUE or CX_FALSE) the
CxSendMechanismErrorAction() CODE API function needs to be called from the client
application before any additional moves can be loaded into the motion queue.

Adding Driver Modules and Libraries to the Makefile

After the servo and I/O interface functions have been developed, a new executable CIMServer must be
generated and linked with these new modules and/or libraries. To accomplish this, the makefile in the
$ROBTOP/lib/cimetrix/custom directory must be modified as follows:

Source Modules

If the servo driver and I/O functions are integrated into a single module or a group of modules, find the
following lines in the makefile:

# Add any new modules to the end of this list.
#Add your custom files here in this section only.
CFILES= PhySignals.c robmain.c threadmain.c user_kin.c \
       user_motion.c user_cntrl.c
CXXFILES=

Add the names of the source modules directly to this list.
Libraries

If the servo driver and I/O functions are integrated in object libraries, find the following lines in the makefile:

```
# Any user libraries which need to be linked with the CIMServer.
USER_LIBS=

ROBLINE_XS=-Xcx -lStr
ROBLIBS=-luserkin -lcomm -lrobutil
NTLIBS= $(USER_LIBS) -lntserver -lserver $(ROBLIBS)
TLIBS= $(USER_LIBS) -lserver -lserver -lserver $(ROBLIBS)
```

Add the appropriate libraries to the `USER_LIBS` definition.

Example

The following code module defines the interface between the CIMServer and the Cimetrix OAC libraries used to interface with the Delta Tau PMAC card. This code provides an example of how a controller interface can be implemented.

```
/***************** Copyright (C) 1995, CIMETRIX Inc. *******************/

FILE rob_to_pmac.c - Interface to PMAC driver interface functions

PUBLIC FUNCTIONS
  pmac_motion_parm
  pmac_motion_func
  pmac_signal_func
  pmac_signal_driver_func
  pmac_send_cmd

PRIVATE FUNCTIONS

REMARKS
  This module contains source code for the pmac driver interface hooks
to the Robline Server

*********************** IMPORTS ***********************

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#ifdef WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
#include <netinet/in.h>
#include <code/const.h>
```
### Forward Declarations

### Global Variables

### Private Data & Variables

### Public Functions

**REMARKS**  
`pmac_motion_parm` - Interface to set/get motion parameters.

This function supports the following functions:
- `CX_CNTRL_GET_CURRENT_POSITION`
- `CX_CNTRL_GET_LATCHED_POSITION`
- `CX_CNTRL_GET_MOTION_STATUS`
- `CX_CNTRL_GET_CONTROLLER_STATUS`
- `CX_CNTRL_GET_CONTROLLER_ATTRIBUTES`
- `CX_CNTRL_GET_ERROR_STATUS`
- `CX_CNTRL_GET_MECHANISM_CNTRL`

**Functions Called**
long pmac_motion_parm ( mechanism_entry *mech, long parm_type, parm_data *parm, 
        CxErrorMsg *this_error )
{
    long err=CX_OK ;
    long halt_status, block_flg ;
    mech_info *minfo ;
    card_info *tmp_card ;
    CxController new_cntrl ;

    minfo = ( mech_info * ) mech->cntrl_info ;
    if ( !minfo && parm_type != CX_CNTRL_GET_CONTROLLER_ATTRIBUTES )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorText ( this_error, "Invalid pointer in pmac_motion_parm function
in PMAC driver" ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
        return ( err ) ;
    }

    switch ( parm_type )
    {
    case CX_CNTRL_GET_CURRENT_POSITION:
        err = oac_get_pos_ptr ( minfo, parm->parm_union.dptr, this_error ) ;
        if ( err == CX_ERROR )
        {
            SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
        }
        break ;

    case CX_CNTRL_GET_LATCHED_POSITION:
        err = oac_get_latch_values ( minfo, parm->parm_union.dptr, this_error ) ;
        if ( err == CX_ERROR )
        {
            SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
        }
        break ;

    case CX_CNTRL_GET_MOTION_STATUS:
        err = oac_is_moving ( minfo, this_error ) ;
        if ( err != CX_ERROR )
        {
            if ( err )
                ...
{  
    parm->parm_union.lval = CX_IN_MOTION ;
}  
else {
    parm->parm_union.lval = CX_NOT_IN_MOTION ;
}
}
else {
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
}
break ;

case CX_CNTRL_GET_CONTROLLER_STATUS:
    mech->controller_stat &= ( CX_MECHANISM_HALTED |  
        CX_KEY_SWITCH_ACTIVE | 
        CX_AMPS_ENABLED | 
        CX_MECHANISM_HOMED ) ;

    if ( mech->controller_stat & CX_AMPS_ENABLED )
    {
        err = oac_check_if_amps_enabled ( minfo, this_error ) ;
        if ( err == CX_ERROR )
        {
            if ( this_error->error == OAC_AMPS_DISABLED )
            {
                mech->controller_stat &= ~CX_AMPS_ENABLED ;
                err = CX_OK ;
            }
            else {
                SetErrorMech ( this_error, mech->client_mech, -1, 
                    CX_FALSE ) ;
                return ( err ) ;
            }
        }
    }

    if ( mech->controller_stat & CX_MECHANISM_HOMED )
    {
        if ( ! oac_get_status_homed ( minfo, this_error ) )
        {
            mech->controller_stat &= ~CX_MECHANISM_HOMED ;
        }
    }

    if ( mech->controller_stat & CX_KEY_SWITCH_ACTIVE )
    {
        if ( !oac_get_teach_key_input ( minfo, this_error ) )
        {
            mech->controller_stat &= ~CX_KEY_SWITCH_ACTIVE ;
        }
    }
}
if ( mech->controller_stat & CX_MECHANISM_HALTED )
{
    err = oac_get_halt_status ( minfo, &halt_status, this_error ) ;
    if ( err == CX_ERROR )
    {
        SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
        return ( err ) ;
    }
    else {
        if ( halt_status == OAC_NOT_HALTED )
        {
            mech->controller_stat &= ~CX_MECHANISM_HALTED ;
        }
    }
}
break ;

case CX_CNTRL_GET_CONTROLLER_ATTRIBUTES:
    mech->controller_attr = CX_TRAJECTORY_MODE |
                          CX_FWDKIN_RATE_MODE |
                          CX_FWDKIN_TIME_MODE |
                          CX_FWDKIN_BLEND_MODE ;
break ;

case CX_CNTRL_SET_OVERRIDE_SPEED:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_SET_OVERRIDE_SPEED case is not
implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
break ;

case CX_CNTRL_SET_BLEND_TOLERANCE:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_SET_BLEND_TOLERANCE case is not
implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
break ;

case CX_CNTRL_SET_TARGET_POLLING_FREQ:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_SET_TARGET_POLLING_FREQ case is
not implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
break ;
case CX_CNTRL_SET_CNTRL_TRAJ_RATE:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_SET_CNTRL_TRAJ_RATE case is not implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
    break ;

case CX_CNTRL_SET_HOME_POSITION:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_SET_HOME_POSITION case is not implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
    break ;

case CX_CNTRL_GET_ERROR_STATUS:
    block_flg = parm->parm_union.lval ;
    if ( block_flg )
    {
        err = oac_monitor_errors ( minfo, this_error ) ;
        if ( err == CX_ERROR )
        {
            SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
        }
    }
    break ;

case CX_CNTRL_GET_MECHANISM_CNTRL:
    tmp_card = get_card_from_mech ( minfo, this_error ) ;
    if ( !tmp_card )
    {
        err = CX_ERROR ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    }
    else {
        new_cntrl = ( CxController ) parm->parm_union.vptr ;
        new_cntrl->driver_info = ( caddr_t ) tmp_card ;
    }
    break ;

default:
    SetErrorMsg ( this_error, CX_INVALID_PARM_TYPE ) ;
    SetErrorText ( this_error, "Invalid parm_type for the pmac_motion_parm function in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    err = CX_ERROR ;
    break ;
long pmac_motion_func ( mechanism_entry *mech, long function, void *arg, 
CxErrorMsg *this_error )
{
    long err ;
    mech_info *minfo ;
    long open_mode, enable_flg, block_flg ;
    caddr_t tmpinfo ;
    command_entry *q ;
    double jnt_time ;

    minfo = ( mech_info * ) mech->cntrl_info ;
    switch ( function )
    {
        case CX_CNTRL_STOP_MOTION:
            block_flg = ( long ) arg ;
            if ( block_flg )
            {
                if ( !minfo )
                {
                    err = CX_ERROR ;
                    // Handle error
                }
            }
            else
            {
                // Handle non-blocking case
            }
            break ;
        // Handle other functions
    }
}

REMARKS    pmac_motion_func - Interface to PMAC motion related functions
This function supports the following functions:
    CX_CNTRL_STOP_MOTION
    CX_CNTRL_WAIT_FOR_MOTION_STOP
    CX_CNTRL_CLOSE_CONTROLLER
    CX_CNTRL_AMPLIFIER_CONTROL
    CX_CNTRL_INITIATE_HOME_SEQUENCE
    CX_CNTRL_INVOKE_TEACH_PENDANT
    CX_CNTRL_INVOKE_TELE_MASTER
    CX_CNTRL_OPEN_CONTROLLER
    CX_CNTRL_FLUSH_MOTION_QUEUE
    CX_CNTRL_RESUME_MOTION
    CX_CNTRL_SEND_MOVE
    CX_CNTRL_INITIALIZE_CONTROLLER

FUNCTIONS CALLED

WRITTEN BY

MODIFICATION LOG

long pmac_motion_func ( mechanism_entry *mech, long function, void *arg, 
CxErrorMsg *this_error )
{
Customizing CODE

```c
SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
SetErrorText ( this_error, "Invalid pointer in
  CX_CNTRL_STOP_MOTION case in
  pmac_motion_func function in PMAC driver"
) ;
SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE )
;
}
else {
  err = oac_stop ( minfo, this_error ) ;
  if ( err == CX_ERROR )
  {
    SetErrorMech ( this_error, mech->client_mech, -1,
      CX_TRUE ) ;
  }
}
}
else {
  err = CX_ERROR ;
  SetErrorMsg ( this_error, CX_NO_POLL_STOP_MOTION_ROUTINE ) ;
  SetErrorText ( this_error, "No POLLING for
  CX_CNTRL_STOP_MOTION in PMAC driver" ) ;
  SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
}
break ;

case CX_CNTRL_WAIT_FOR_MOTION_STOP:
  block_flg = ( long ) arg ;
  if ( block_flg )
  {
    if ( !minfo )
    {
      err = CX_ERROR ;
      SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
      SetErrorText ( this_error, "Invalid pointer in
        CX_CNTRL_WAIT_FOR_MOTION_STOP case in
        pmac_motion_func function in PMAC driver"
) ;
      SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE )
    }
  }
  else {
    err = oac_block ( minfo, this_error ) ;
    if ( err == CX_ERROR )
    {
      if ( this_error->error == DOAC_RESET_ABORT )
      {
        err = CX_OK ;
      }
      else {
        // More error handling code here...
      }
    }
```
SetErrorMech ( this_error, mech->client_mech, -1,
             CX_TRUE ) ;
}
}
}
}
}
}
} else {
    err = CX_ERROR ;
    SetErrorMsg ( this_error, CX_NO_POLL_WAIT_MOTION_STOP_ROUTINE ) ;
    SetErrorMsg ( this_error, "No POLLING for
             CX_CNTRL_WAIT_FOR_MOTION_STOP in PMAC
             driver" ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
} break ;

case CX_CNTRL_CLOSE_CONTROLLER:
    if ( !minfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorMsg ( this_error, "Invalid pointer in
                   CX_CNTRL_CLOSE_CONTROLLER case in
                   pmac_motion_func function in PMAC driver" ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    }
    else {
        oac_abort_cs ( minfo, this_error ) ;
        oac_close ( minfo, this_error ) ;
        mech->cntrl_info = ( caddr_t ) NULL ;
    } break ;

case CX_CNTRL_AMPLIFIER_CONTROL:
    if ( !minfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorMsg ( this_error, "Invalid pointer in
                   CX_CNTRL_AMPLIFIER_CONTROL case in
                   pmac_motion_func function in PMAC driver" ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    }
    else {
        enable_flg = ( long ) arg ;
        if ( enable_flg )
        {
            err = oac_enable_amps ( minfo, 1, this_error ) ;
        }
if ( err == CX_ERROR )
{
    SetErrorMech ( this_error, mech->client_mech, -1,
                  CX_TRUE ) ;
}
} else {
    err = oac_disable_amps ( minfo, this_error ) ;
    if ( err == CX_ERROR )
    {
        SetErrorMech ( this_error, mech->client_mech, -1,
                       CX_TRUE ) ;
    }
}
}
break ;

case CX_CNTRL_INITIATE_HOME_SEQUENCE:
    if ( !minfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorText ( this_error, "Invalid pointer in
                       CX_CNTRL_AMPLIFIER_CONTROL case in
                       pmac_motion_func function in PMAC driver"
                      ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    } else {
        err = oac_send_home ( minfo, DRVR_MOVE_WAIT, this_error ) ;
        if ( err == CX_ERROR )
        {
            if ( this_error->error == DOAC_RESET_ABORT )
            {
                err = CX_OK ;
            } else {
                SetErrorMech ( this_error, mech->client_mech, -1,
                              CX_TRUE ) ;
            }
        }
    }
break;

case CX_CNTRL_INVOKE_TEACH_PENDANT:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_CNTRL_INVOKE_TEACH_PENDANT case is
                     not implemented in the PMAC driver."
                   ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    err = CX_ERROR ;
break ;

case CX_CNTRL_INVOKE_TELE_MASTER:
    SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
    SetErrorText ( this_error, "The CX_CNTRL_INVOKE_TELE_MASTER case is not implemented in the PMAC driver." ) ;
    SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    err = CX_ERROR ;
break ;

case CX_CNTRL_OPEN_CONTROLLER:
    open_mode = ( long ) arg ;
    tmpinfo = ( caddr_t ) oac_open ( mech->rob->pnode->name, open_mode, this_error ) ;
    if ( !tmpinfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorText ( this_error, "Invalid pointer in CX_CNTRL_FLUSH_MOTION_QUEUE case in pmac_motion_func function in PMAC driver" ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    }
    else {
        mech->cntrl_info = tmpinfo ;
        err = CX_OK ;
    }
break ;

case CX_CNTRL_FLUSH_MOTION_QUEUE:
    if ( !minfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetErrorText ( this_error, "Invalid pointer in CX_CNTRL_FLUSH_MOTION_QUEUE case in pmac_motion_func function in PMAC driver" ) ;
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
    }
    else {
        err = oac_restart ( minfo, this_error ) ;
        if ( err == CX_ERROR )
        {
            SetErrorMsg ( this_error, CX_CASE_NOT_IMPLEMENTED ) ;
            SetErrorText ( this_error, "The CX_CNTRL_INVOKE_TELE_MASTER case is not implemented in the PMAC driver." ) ;
            SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
        }
    }
break ;

case CX_CNTRL_RESUME_MOTION:
    if ( !minfo )
    {

err = CX_ERROR;
SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
SetErrorText ( this_error, "Invalid pointer in
    CX_CNTRL_RESUME_MOTION case in
    pmac_motion_func function in PMAC driver" ) ;
SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
}
else {
    err = oac_resume ( minfo, this_error ) ;
    if ( err == CX_ERROR )
    {
        SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE )
    }
} else {
    q = (command_entry *)arg ;
    switch ( q->trajectory_mode )
    {
    case CX_END_POINT_MOVE:
        jnt_time = estimate_jnt_time ( q, -1 ) ;
        if(jnt_time < CX_EPSILON)
        {
            err = CX_OK;
            break;
        }
        err = oac_set_acc ( minfo, q->ramp_time_rise, q->
            >rise_S_time, CX_MOVE_THRU, this_error ) ;
        if ( err == CX_ERROR )
        {
            if ( this_error->error == DOAC_RESET_ABORT )
            {
                err = CX_OK ;
            }
        }
    case CX_CNTRL_SEND_MOVE:
        if ( !minfo )
        {
            err = CX_ERROR ;
            SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
            SetErrorText ( this_error, "Invalid pointer in
                CX_CNTRL_SEND_MOVE case in
                pmac_motion_func function in PMAC driver" ) ;
            SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
        }
        else {
            q = (command_entry *)arg ;
            switch ( q->trajectory_mode )
            {
            case CX_END_POINT_MOVE:
                jnt_time = estimate_jnt_time ( q, -1 ) ;
                if(jnt_time < CX_EPSILON)
                {
                    err = CX_OK;
                    break;
                }
                err = oac_set_acc ( minfo, q->ramp_time_rise, q->
                    >rise_S_time, CX_MOVE_THRU, this_error ) ;
                if ( err == CX_ERROR )
                {
                    if ( this_error->error == DOAC_RESET_ABORT )
                    {
                        err = CX_OK ;
                    }
                }
            }
else {
    SetErrorMech ( this_error, mech->client_mech, -1,
        CX_TRUE ) ;
}
break ;
}
err = oac_set_time_move ( minfo, jnt_time, CX_MOVE_THRU,
    this_error ) ;
if ( err == CX_ERROR )
{
    if ( this_error->error == DOAC_RESET_ABORT )
    {
        err = CX_OK ;
    }
    else {
        SetErrorMech ( this_error, mech->client_mech, -1,
            CX_TRUE ) ;
    }
    break ;
}
/* Send Move command to card */
err = oac_send_move ( minfo, q->dof_act, q->blend_policy,
    this_error ) ;
if ( err == CX_ERROR )
{
    if ( this_error->error == DOAC_RESET_ABORT )
    {
        err = CX_OK ;
    }
    else {
        SetErrorMech ( this_error, mech->client_mech, -1,
            CX_TRUE ) ;
    }
    break ;
}
break;
default:
    err = CX_ERROR;
    SetErrorMsg (this_error, CX_UNDEFINED_MOVE_MODE);
    SetErrorMech (this_error, mech->client_mech, -1, CX_TRUE);
    SetErrorText (this_error, "Undefined trajectory mode for
    CX_CNTRL_SEND_MOVE in PMAC driver");
    break;
}

break;

case CX_CNTRL_INITIALIZE_CONTROLLER:
    if (!minfo)
    {
        err = CX_ERROR;
        SetErrorMsg (this_error, CX_NIL_PTR_REFERENCED);
        SetErrorText (this_error, "Invalid pointer in
        CX_CNTRL_INITIALIZE_CONTROLLER case in
        pmac_motion_func function in PMAC driver");
        SetErrorMech (this_error, mech->client_mech, -1, CX_TRUE);
    }
    else {
        if (oac_set_move_mode (minfo, MMODE_LINEAR, CX_MOVE_WAIT, this_error) == CX_ERROR)
        {
            err = CX_ERROR;
            SetErrorMech (this_error, mech->client_mech, -1, CX_TRUE);
            return (err);
        }
        if (oac_set_move_type (minfo, MTYPE_ABS, CX_MOVE_WAIT, this_error) == CX_ERROR)
        {
            err = CX_ERROR;
            SetErrorMech (this_error, mech->client_mech, -1, CX_TRUE);
            return (err);
        }
        if (oac_set_acc (minfo, 0.2, 0.0, CX_MOVE_WAIT, this_error) == CX_ERROR)
        {
            err = CX_ERROR;
            SetErrorMech (this_error, mech->client_mech, -1, CX_TRUE);
            return (err);
        }
    }
    break;
default:
  SetErrorMsg ( this_error, CX_INVALID_FUNC_TYPE ) ;
  SetErrorText ( this_error, "Invalid function type for the
  pmac_motion_func function in the PMAC
driver." ) ;
  SetErrorMech ( this_error, mech->client_mech, -1, CX_TRUE ) ;
  err = CX_ERROR ;
  break ;
}
return ( err ) ;
}

/*---------------------------------------------------------------------
REMARKS pmac_signal_func - Interface to signal relation PMAC functions.
This function supports the following functions:
  CX_CNTRL_INITIALIZE_SIGNAL
  CX_CNTRL_GET_SIGNAL_VALUE
  CX_CNTRL_SET_LATCH
  CX_CNTRL_SET_SIGNAL_VALUE
  CX_CNTRL_SETUP_SIGNAL_MONITORING
  CX_CNTRL_GET_SIGNAL_CNTRL
FUNCTIONS CALLED
WRITTEN BY
modification log
---------------------------------------------------------------------*/
long pmac_signal_func ( signals_t *sigPtr, long function, void *arg, CxErrorMsg
  *this_error )
{
  long err=CX_OK ;
  mech_info *minfo ;
  mechanism_entry *mech ;
  signal_info *sig_info ;
  CxController new_cntrl ;

  switch ( function )
  {
  case CX_CNTRL_INITIALIZE_SIGNAL:
    err = oac_initialize_signal ( sigPtr, this_error ) ;
    break ;

  case CX_CNTRL_GET_SIGNAL_VALUE:
    err = oac_get_signal ( sigPtr, ( long * ) arg, this_error ) ;
    break ;
case CX_CNTRL_PULSE_SIGNAL:
    break ;

case CX_CNTRL_SET_LATCH:
    sig_info = ( signal_info * ) arg ;
    mech = ( mechanism_entry * ) sig_info->mech ;
    minfo = ( mech_info * ) mech->cntrl_info ;
    if ( !minfo )
    {
        err = CX_ERROR ;
        SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
        SetLastErrorText ( this_error, "Invalid pointer in
                        CX_CNTRL_SET_LATCH case in
                        pmac_signal_func function in PMAC driver" ) ;
        SetLastErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
    } else {
        err = oac_set_latch ( minfo, sigPtr, sig_info->change, this_error);
    }
    break ;

case CX_CNTRL_SET_SIGNAL_VALUE:
    err = oac_set_signal ( sigPtr, ( long ) arg, this_error ) ;
    break ;

case CX_CNTRL_SETUP_SIGNAL_MONITORING:
    sig_info = ( signal_info * ) arg ;
    if ( sig_info->mech )
    {
        minfo = ( mech_info * ) sig_info->mech->cntrl_info ;
        if ( !minfo )
        {
            err = CX_ERROR ;
            SetErrorMsg ( this_error, CX_NIL_PTR_REFERENCED ) ;
            SetLastErrorText ( this_error, "Invalid pointer in
                            CX_CNTRL_SETUP_SIGNAL_MONITORING case in
                            pmac_signal_func function in PMAC driver" ) ;
            SetLastErrorMech ( this_error, mech->client_mech, -1, CX_FALSE ) ;
            return ( err ) ;
        }
    } else {
        minfo = ( mech_info * ) NULL ;
    }
err = oac_setup_signal_polling ( sigPtr, sig_info->comparison,
    sig_info->value, sig_info->operation,
    minfo, this_error ) ;

break ;

case CX_CNTRL_GET_SIGNAL_CNTRL:
    new_cntrl = ( CxController ) arg ;
    new_cntrl->driver_info = ( caddr_t ) get_card_from_signal ( sigPtr,
        this_error ) ;
    break ;

default:
    SetErrorMsg ( this_error, CX_INVALID_FUNC_TYPE ) ;
    SetErrorText ( this_error, "$Invalid function type for the
    pmac_signal_func function in the PMAC
driver." ) ;
    err = CX_ERROR ;
    break ;
}

return ( err ) ;
}

/*---------------------------------------------------------------------
REMARKS    pmac_signal_drvr_func - Interface to signal driver functions.

This function supports the following functions:

CX_CNTRL_SET_SIG_SCAN_RATE
CX_CNTRL_CLOSE_SIGNAL_TABLE
CX_CNTRL_MONITOR_SIGNALS
CX_CNTRL_OPEN_SIGNAL_TABLE

FUNCTIONS CALLED

WRITTEN BY
    Peter Manley                                 March, April - 1995

MODIFICATION LOG

---------------------------------------------------------------------*/
long pmac_signal_drvr_func ( signal_drivers_t *drvrPtr, long function,
   void *arg, CxErrorMsg *this_error )
{
    long err ;
    signal_info *sigInfo ;

    switch ( function )
    {
    case CX_CNTRL_SET_SIG_SCAN_RATE:
break ;

case CX_CNTRL_CLOSE_SIGNAL_TABLE:
    drvrPtr->initialized = CX_FALSE ;
    err = oac_close_signal_table ( this_error ) ;
    break ;

case CX_CNTRL_MONITOR_SIGNALS:
    sigInfo = ( signal_info * ) arg ;
    err = oac_monitor_signals ( &sigInfo->sigPtr, &sigInfo->value,
                   this_error ) ;
    break ;

case CX_CNTRL_OPEN_SIGNAL_TABLE:
    err = oac_open_signal_table ( this_error ) ;
    if ( err != CX_ERROR )
    {
        drvrPtr->initialized = CX_TRUE ;
    }
    break ;

default:
    SetErrorMsg ( this_error, CX_INVALID_FUNC_TYPE ) ;
    SetErrorText ( this_error, "Invalid function type for the
                 pmac_signal_drvr_func function in the
                 PMAC driver." ) ;
    err = CX_ERROR ;

    break ;
}
return ( err ) ;
}

/*---------------------------------------------------------------------
REMARKS   pmac_send_cmd - Interface to general PMAC commands not supported
by the CIMServer.
FUNCTIONS CALLED
WRITTEN BY
MODIFICATION LOG

---------------------------------------------------------------------*/

long pmac_send_cmd ( cmd_msg *msg, CxErrorMsg *this_error )
{
    long err=CX_OK ;
    unsigned char *tmpStrPtr, *sStrPtr, *tmpBuf ;
    unsigned long tmpLong ;
long ivar, i, version;
double value;
char operation, tmpChar[8];
card_info *cinfo;
signals_t **gather_sig;
long num_sig, icnt, period, mode, cnt, buff_size;
char *format;
long trigger;
char *buff=NULL, *curPtr;
long size_of_buff, act_cnt, start_cnt, num_cnt, num_packet;
cmd_msg upload_packet;

tmpStrPtr = sStrPtr = ( unsigned char * ) msg->cmd ;
memcpy ( ( void* ) &operation, ( void * ) tmpStrPtr, sizeof ( operation ) )
	;
tmpStrPtr += 1 ;

switch ( operation )
{
    case PMAC_S_PMAC_IVAR :
        memcpy( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof( tmpLong ) );
        ivar = ( long ) ntohl ( ( unsigned long ) tmpLong ) ;
        tmpStrPtr += 4 ;

        tmpBuf = ( unsigned char * ) &value ;
        memcpy( ( void * ) tmpBuf, ( void * ) tmpStrPtr, 8 ) ;
        #if defined LYNX && defined I386
        memcpy( ( void * ) tmpChar, ( void * ) tmpBuf, 8 ) ;
        for ( i = 0 ; i < 8 ; i++ )
            tmpBuf[i] = tmpChar[7-i] ;
        #endif
        tmpStrPtr += 8 ;

        cinfo = ( card_info * ) msg->driver_info ;

        err = set_pmac_ivar ( cinfo, ivar, value, this_error ) ;
        break ;

    case PMAC_Q_PMAC_IVAR :
        memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
        ivar = ( long ) ntohl ( ( unsigned long ) tmpLong ) ;
        tmpStrPtr += 4 ;

        cinfo = ( card_info * ) msg->driver_info ;
err = get_pmac_ivar ( cinfo, ivar, &value, this_error )
if ( err != CX_ERROR )
{
    tmpStrPtr = sStrPtr = ( unsigned char * ) msg->cmd ;
    tmpBuf = ( unsigned char * ) &value ;
#if defined LYNX && defined I386
    memcpy ( ( void * ) tmpChar, ( void * ) tmpBuf, 8 ) ;
    for ( i = 0 ; i < 8 ; i++ )
    {
        tmpBuf[i] = tmpChar[7-i] ;
    }
#endif
    memcpy ( ( void * ) tmpStrPtr, ( void * ) tmpBuf, 8 ) ;
    tmpStrPtr += 8 ;
    memcpy ( ( void * ) tmpStrPtr, ( void * ) &tmpLong, sizeof ( tmpLong ) ) ;
    tmpStrPtr += 4 ;
    msg->cmd_len = tmpStrPtr - sStrPtr ;
}
break ;

case PMAC_GET_VERSION:
    cinfo = ( card_info * ) msg->driver_info ;
    version = oac_get_interconnect_version ( cinfo, this_error ) ;
    tmpStrPtr = sStrPtr = ( unsigned char * ) msg->cmd ;
    tmpLong = ( unsigned long ) htonl ( ( unsigned long ) version ) ;
    memcpy ( ( void * ) tmpStrPtr, ( void * ) &tmpLong, sizeof ( tmpLong ) ) ;
    tmpStrPtr += 4 ;
    msg->cmd_len = tmpStrPtr - sStrPtr ;
    break ;

case PMAC_GATH_INIT:
    memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
    num_sig = ( long ) ntohl ( ( unsigned long ) tmpLong ) ;
    tmpStrPtr += 4 ;
    gather_sig = ( signals_t ** ) calloc ( ( size_t ) num_sig, sizeof ( signals_t * ) ) ;
    if ( gather_sig )
    {
        for ( i = 0 ; i < num_sig ; i++ )
        {
            memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
        }
icnt = (long) ntohl((unsigned long) tmpLong);
gather_sig[i] = signals[icnt];
tmpStrPtr += 4;
}

memcpy((void*) &tmpLong, (void*) tmpStrPtr, sizeof(tmpLong));
period = (long) ntohl((unsigned long) tmpLong);
tmpStrPtr += 4;

memcpy((void*) &tmpLong, (void*) tmpStrPtr, sizeof(tmpLong));
mode = (long) ntohl((unsigned long) tmpLong);
tmpStrPtr += 4;

memcpy((void*) &tmpLong, (void*) tmpStrPtr, sizeof(tmpLong));
cnt = (long) ntohl((unsigned long) tmpLong);
tmpStrPtr += 4;

format = (char*) calloc((size_t)((num_sig * 5)),
sizeof(char));
if(format)
{
    /* call gather_init here */
    
    err = gather_init(num_sig, gather_sig, period, mode,
                        &cnt, &buff_size, format, this_error);
    if(err == CX_ERROR)
    {
        free((char*) gather_sig);
        free((char*) format);
        return(err);
    }
    free((char*) gather_sig);

    tmpStrPtr = sStrPtr = (unsigned char*) msg->cmd;

tmpLong = (unsigned long) htonl((unsigned long) cnt);
memcpy((void*) tmpStrPtr, (void*) &tmpLong, sizeof(tmpLong));
tmpStrPtr += 4;

tmpLong = (unsigned long) htonl((unsigned long) buff_size);
memcpy((void*) tmpStrPtr, (void*) &tmpLong, sizeof(tmpLong));
tmpStrPtr += 4;
tmpLong = htonl ( strlen ( format ) ) ;
memcpy ( ( void * ) tmpStrPtr, ( void * ) &tmpLong, sizeof ( tmpLong ) ) ;
tmpStrPtr += 4 ;
memcpy ( ( void * ) tmpStrPtr, ( void * ) format, strlen ( format ) ) ;
tmpStrPtr += strlen ( format ) ;
free ( ( char * ) format ) ;
msg->cmd_len = tmpStrPtr - sStrPtr ;
}
else {
    err = CX_ERROR ;
    SetErrorMsg ( this_error, CX_MACHINE_OUT_OF_MEMORY ) ;
    if ( gather_sig )
        free ( ( char * ) gather_sig ) ;
}
}
else {
    err = CX_ERROR ;
    SetErrorMsg ( this_error, CX_MACHINE_OUT_OF_MEMORY ) ;
}
break ;
case PMAC_GATH_START :
    memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
    trigger = ntohl ( ( unsigned long ) tmpLong ) ;
tmpStrPtr += 4 ;
    err = gather_start ( trigger, this_error ) ;
    break ;
case PMAC_GATH_STOP :
    err = gather_stop ( this_error ) ;
    break ;
case PMAC_GATH_UPLOAD :
    memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
    start_cnt = ntohl ( ( unsigned long ) tmpLong ) ;
tmpStrPtr += 4 ;
    memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
num_cnt = ( long ) ntohl ( ( unsigned long ) tmpLong ) ;
tmpStrPtr += 4 ;

memcpy ( ( void * ) &tmpLong, ( void * ) tmpStrPtr, sizeof ( tmpLong ) ) ;
size_of_buff = ( long ) ntohl ( ( unsigned long ) tmpLong ) ;
tmpStrPtr += 4 ;

buff = ( char * ) malloc ( size_of_buff ) ;
if ( buff ) {
    /* call gather_upload here */

    err = gather_upload_card ( start_cnt, num_cnt, &act_cnt, buff, this_error ) ;
    if ( err == CX_ERROR ) {
        free ( ( char * ) buff ) ;
        return ( err ) ;
    } else {
        tmpStrPtr = sStrPtr = ( unsigned char * )
        &upload_packet.cmd ;

        tmpLong = ( unsigned long ) htonl ( ( unsigned long )
        act_cnt ) ;
        memcpy ( ( void * ) tmpStrPtr, ( void * ) &tmpLong, sizeof ( tmpLong ) ) ;
        tmpStrPtr += 4 ;

        num_packet = size_of_buff/CX_MAX_CMD_STR + 1 ?
        size_of_buff/CX_MAX_CMD_STR + 1:
        size_of_buff/CX_MAX_CMD_STR ;

        tmpLong = ( unsigned long ) htonl ( ( unsigned long )
        num_packet ) ;
        memcpy ( ( void * ) tmpStrPtr, ( void * ) &tmpLong, sizeof ( tmpLong ) ) ;
        tmpStrPtr += 4 ;

        curPtr = buff ;
        upload_packet.cmd_len = 8 ;
        upload_packet.mtype = CX_SEND_DEVICE_CMD_REPLY ;
        upload_packet.ret_code = CX_OK ;
        if ( CxMSendCmdMsg ( msg->client_on_host, &upload_packet, sizeof ( cmd_msg ) ) == CX_ERROR ) {
            /* Do error handling */
        }
    }
}
fprintf ( stderr, ”ERROR - unable to send SEND_DEVICE_CMD_REPLY to client\n” ) ;
}
upload_packet.mtype = CX_GATHERED_DATA_REPLY ;
upload_packet.ret_code = CX_OK ;
for ( i = 0 ; i < num_packet ; i++ )
{
    upload_packet.ret_code = 0 ;
    if ( i < num_packet-1 )
    {
        upload_packet.cmd_len = CX_MAX_CMD_STR ;
        memcpy ( upload_packet.cmd, curPtr, upload_packet.cmd_len ) ;
        curPtr += upload_packet.cmd_len ;
    }
    else
    {
        upload_packet.cmd_len = size_of_buff - i*CX_MAX_CMD_STR ;
        memcpy ( upload_packet.cmd, curPtr, upload_packet.cmd_len ) ;
    }
    if ( CxMSendCmdMsg ( msg->client_on_host, &upload_packet, sizeof ( cmd_msg ) ) == CX_ERROR )
    {
        fprintf ( stderr, ”ERROR - unable to send SEND_DEVICE_CMD_REPLY to client\n” ) ;
    }
    free ( ( char * ) buff ) ;
}
else  {
    err = CX_ERROR ;
    SetErrorMsg ( this_error, CX_MACHINE_OUT_OF_MEMORY ) ;
}
break ;
default:
    err = CX_ERROR ;
    SetErrorMsg ( this_error, CX_UNDEFINED_CMD_FUNC ) ;
    break ;
}
return ( err ) ;
Chapter 5

Custom Motion Interfaces

Introduction

The development of custom trajectory generators for physical control of mechanisms must be approached with caution! This chapter cannot act as a substitute for actual control experience. It is thus requested that you approach the implementation of custom routines with great care. Since the CIMServer provides a simulation capability, you should first simulate your trajectory algorithms before using them to drive physical mechanisms. Examination of the joint values, accelerations, speeds, and time increments can also be useful since you have access to the data structure members in your custom routines.

This chapter outlines the interfaces provided to the user to control the motion of various mechanisms. This material is concerned with control using an open architecture controller, such as the Cimetrix controller, where the CIMServer’s motion control interfaces can be used for trajectory control; thus, it does not apply to control of existing mechanism controllers where the interface permits macro-level motion commands to be issued via a device interface to the controller.

Since, as discussed in Chapter 0 on CODE Kinematics, a robot or similar mechanism such as a machine tool is a collection of links and joints, motion control is concerned with the way joints are moved between specified robot configurations. Because many commercial robots have secondary dependent linkages (e.g., actuators) which serve to actuate the independent joints, interfaces are provided for the user to control both dependent joint actuators and independent joint actuators. Dependent joints serve to actuate an independent joint to a specified joint value. An example of a set of dependent joints is a ball screw and associated four bar linkage used to rotate a primary robot link about a revolute joint.

Robots can be reconfigured in one of two ways: by specifying the joint angles for a new robot configuration (forward kinematics) or by specifying a desired Cartesian reference position and orientation (pose) of some tool attached to the end of the robot (as defined by a tool or terminal control frame, or simply TCF), and then applying a solution set to determine the joint angles necessary to move the TCF to the desired pose (inverse kinematics). In either case the user can move between the configurations by joint control or Cartesian control.

Of course, all motion control eventually resorts to joint control in the servo cycle. In Cartesian control a trajectory generator continually samples the desired Cartesian path at a trajectory rate along the path.
between the desired robot configurations. Inverse kinematics are then used to calculate, at the trajectory rate, the necessary joint changes and joint rates to maintain the Cartesian path motion. When fed to the motion control card (or servo card), internal joint control algorithms blend the joint motion according to the joint changes specified. In the servo card the blending of motion occurs at a rate which is typically one or two orders-of-magnitude faster than the trajectory rate. The trajectory rate is typically 20 - 200 Hz, depending upon the mechanism complexity.

CODE motion control between configurations can use either joint control algorithms or Cartesian trajectory algorithms to control curvilinear motion. The following sections outline the open architecture interfaces provided the user to control motion in these various modes.

**CODE Trajectory Control**

Trajectory control is the planning and conduct of incremental motions along a desired trajectory and at rates and accelerations that fall within the physical capabilities of the mechanism. Two typical control modes are used: joint interpolation and Cartesian path following. Joint interpolated motion is coordinated control in joint space whereby the faster joint moves are slowed to the rate of the slowest joint so that all accomplish their joint change in the same time period. A trajectory generator is used to accomplish the motion.

Joint motion (CX_JOINT_INTERP) is the default mode in CODE. The move type can be set by the API CxSetInterpType() and valid values are CX_JOINT_INTERP (to signify a joint interpolated move) and CX_LINEAR_INTERP (to signify a curvilinear path type move). There are other move types, such as CX_CIRCULAR_INTERP, but these are handled internally depending upon the segment type.

**Trajectory Generator**

The default trajectory generator currently implemented in the CIMServer is a simple trapezoidal generator that uses constant accelerations and decelerations to change to the desired speed. See Figure 4-1. The user, through the custom trajectory interfaces, described later in this chapter, may provide custom routines for trajectory control if more sophisticated control is desired.

![Figure 4-1: Trapezoidal Trajectory Generator.](speed_time.png)
If the user specifies move segments that are too short, the trapezoidal form will assume the triangular shape shown in Figure 4-2. The obvious result is that it will not be possible for the move to reach the desired (set) speed.

![Figure 4-2: Speed Reduction for Short Moves.](image)

**Motion Blending**

The blending policy, applied to either joint moves or moves along curvilinear paths, and as set by the API `CxSetBlendPolicy()`, has as its default `CX_MOVE_WAIT`. `CX_MOVE_WAIT` will not queue up move commands, while the `CX_MOVE_TO` policy will immediately load moves into the queue and cause these move functions to return immediately in the CODE application process. Because these policies constrain moves to start and end with zero speed, they constitute what is commonly referred to as point-to-point motion.

The policy `CX_MOVE_THRU` will queue up each move and immediately return from the move API. When two or more moves are queued up, the deceleration period of the first move will be blended (added) to the acceleration period of the next move in the queue. This continuous motion will, in effect, maintain near path speeds in the transition from one move segment to the next at the sacrifice of path accuracy, Figure 4-3. The blending vertex is the end configuration (joint moves) or pose (curvilinear moves) of the first segment and the start configuration or pose of the next move. Speed reduction will be a function of the difference in tangency of the two moves.

The `CX_MOVE_THRU` policy can be used for blending joint moves or moves along a CODE curve (a collection of move segments called a curveseg) or for blending separate, but queued moves of the `CX_LINEAR_INTERP` or `CX_CIRCULAR_INTERP` type.

![Figure 4-3: CX_MOVE_THRU Blending.](image)

For paths defined as a curveseg, blending is not necessary between path segments that are tangent to each other. If the user sets the blend policy to either `CX_MOVE_TO_TANGENT` or `CX_MOVE_THRU_TANGENT`, the CIMServer will perform a tangency check between contiguous...
Customizing CODE

curve segments for a \textit{curveseg} (linked list of curvilinear segments) and then decide whether blending is necessary; otherwise acceleration/deceleration periods are simply used to transition between segment speeds as shown in Figure 4-4. These policies permit holding the exact path, while also maintaining the desired speed along the path. The tangent blend policy can also be applied to a sequence of joint moves which involve a single axis, where tangency exists if the joint sequence is in one joint direction.

The difference between these two blend policies is the way the motion occurs at the non-tangent segments. If the \texttt{CX\_MOVE\_TO\_TANGENT} policy is used, the blend policy will be set to \texttt{CX\_MOVE\_TO} at the non-tangent seg, causing the motion to stop at the end of the non-tangent seg, followed by an acceleration to the desired speed on the seg following the non-tangent seg. If the \texttt{CX\_MOVE\_THRU\_TANGENT} policy is used, the blend policy will be set to \texttt{CX\_MOVE\_THRU} at the non-tangent seg, causing motion between the non_tangent seg and the following seg to be blended.

Joint Motion (\texttt{CX\_JOINT\_INTERP})

Joint motion is the default mode in CODE and is set by the API \texttt{CxSetInterpType()}. Joint changes can be specified directly by commanding a joint configuration, \texttt{CxMoveSingleAxis()} or \texttt{CxMoveAllAxes()}, or as a target move which uses inverse kinematics to determine the joint change necessary to move the current tool TCF (tool control frame) to its pose in the final target configuration. If motion is being controlled by the Cimetrix controller, the user can send the joint changes as the end point moves directly to the servo card, and it will use firmware algorithms to make the joint interpolated move. The user also has the option to use the CIMServer's software algorithms to command the incremental moves in a PVT mode (position-velocity-time) and thereby have software control of the trajectory at the incremental level, but at a coarser trajectory rate.

Joint rate control is specified by the user entering a joint speed, using the API \texttt{CxSetJointSpeed()}, along with the maximum joint rates for each axis, using the API \texttt{CxSetMaxJntSpeed()}. The joint speed is a number between 0 and 1. The default is 0.1 or 10% of full speed.

The CIMServer’s acceleration control currently permits two modes: \texttt{CX\_CONST\_RAMP\_TIME} (time to accomplish the change in speed as specified by the API \texttt{CxSetTrapAccelTimes()}) and \texttt{CX\_CONST\_RAMP\_ACCEL} (acceleration magnitudes directly specified). Both modes depend upon the user to specify the desired maximum acceleration and deceleration values for each joint (\texttt{CxSetJntAccel()}, \texttt{CxSetJntAccelMax()}). The default values are set artificially low in CODE and will cause extremely slow motion on the simulation screen or in actual control. Thus, the user must, with great care, set values appropriate to the mechanism being controlled. Any computed accelerations required to accomplish a desired speed change will be compared to the maximums. If maximums are exceeded, then the trajectory accelerations will be reduced to the maximums.
**WARNING!**: If the user notices slower than expected moves on the
CX_SIMULATION screen or during CX_RUNTIME moves on physical mechanisms,
particularly for translational (sliding) joints, then the acceleration *maximums* are
probably set to the default values and may be too low. A reasonable approximation to
allowable acceleration values can be determined by dividing the desired speed change
by an acceleration time period of 0.1 - 0.5 seconds, but even these accelerations may
be too large for some mechanisms. The user is responsible for determining these
acceleration *maximums* and entering them properly.

CODE permits the acceleration times and magnitudes to vary from the deceleration times and
magnitudes, and also permits these values to change dynamically in the motion control sequence.
CODE allows linear type moves between joint configurations when the tool motion type is set to
CX_LINEAR_INTERP and the API CxMoveAllAxes ( ) or the API CxMoveToConfig) is used.
Inverse kinematics are automatically applied to the last independent joint frame to determine a target in
the desired robot configuration. This assumes that inverse kinematics are available for the robot.

**Cartesian Path Motion**
Cartesian path following uses trajectory algorithms and inverse kinematics to incrementally determine
the joint changes, joint rates, and joint accelerations necessary to accomplish the path following of
Cartesian space paths as lines and arcs. A nominal trajectory generator (like the one used in the
CIMServer) uses a simple trapezoidal profile to accomplish speed changes, with necessary checks to
ensure that joint rates and accelerations are not being exceeded.

The typical trajectory process is shown in the diagram of Figure 4-5. A move buffer sends down moves,
along with the necessary move settings, to the routine user_trajectory ( ), which then calls trajectory
specific routines to initialize the move parameters and make incremental moves along the path segment.
If motion is a Cartesian type move, then inverse kinematics routines are called by the function pick_soln
to get new joint values for the incremental move if the joint rate checking is done in actuator space, then
the joint values and joint rates must first be converted to actuator values (user_convert_jnt_to_act( )).
Rate checking then calls routines to determine the joint rates (and possibly the joint accelerations) at the new configuration and at the desired Cartesian speeds (and possibly the accelerations). If rates are exceeded, then the joint speeds must be reduced to allowable values. Some algorithms call the trajectory generation routines again to modify the move to reflect the decrease in Cartesian speeds that result from the corresponding decrease in joint speeds to allowable limits — see Figure 4-5.

Once the rates are acceptable, move parameters are issued to the Cimetrix controller, if in CX_RUNTIME mode, or to a device driver to an existing controller, or to the simulation screen, if in CX_ANIMATION mode.

Figure 4-5 shows three routines: valid_tcf_target, get_target_frames, and motion_start, embedded in the CIMServer’s default trajectory generator that is used to initialize moves. One of the advantages of the CODE architecture is that target frame information can be extracted from the CIMServer’s Cell Model database dynamically. If the user uses the API CxSetMotionTrack to set motion tracking ON, motion_start() will be called in every trajectory cycle to update parameters such as the desired target frame. This will allow the mechanism to track dynamically or quasi-statically moving targets. The CODE architecture thus permits sensor integration with the Cell Model database, i.e., a sensor reading can be used to update target frame information in real-time. This updated information can be used to deviate the trajectory dynamically.

The user would normally conduct tracking at lower trajectory rates and only when sensor/tracking integration is actually to occur. Tracking should be turned off for non-tracking moves. Tracking, of course, requires that the mechanism be capable of moving at greater speeds than the targets being tracked.

If the default CIMServer trajectory routines are not used, then routines can be developed by the user. Sections which follow describe the interfaces provided for this purpose, along with key members of the mechanism_entry and command_entry data structures.
Maintaining Speeds Along Contiguous Path Segments

To maintain curvilinear speed along path segments for a CX_MOVE_THRU policy, path segments must exceed a certain length if they are not tangent to each other. Otherwise, the robot will simply slow down during these short segments, by virtue of keeping the trajectory time fixed and not being able to attain full speed before the deceleration period is reached.

For non-tangent segments the CIMServer’s trajectory generator must have sufficient time to accelerate to the desired speeds; otherwise, the ramp algorithms will begin the necessary deceleration required for blending before these speeds are reached — refer back to Figure 4-2.

An estimate of the necessary path length follows for each acceleration mode currently used in the CIMServer:

\[
\text{CX CONST RAMP TIME: } \text{length} > \text{speed} \times \text{accel time} / 2
\]

\[
\text{CX CONST RAMP ACCEL: } \text{length} > \frac{\text{speed}^2}{2 \times \text{accel magnitude}}
\]

If a path of contiguous segments is defined as a CODE curveseg (linked list of path segments), and each contiguous pair is mutually tangent, then the path segments can be shorter than the minimum lengths listed previously. The length limitation now becomes a function of the time to transition the segment and whether this time is greater than the trajectory time step (inverse of the trajectory frequency in Hz).

The length can be estimated by the following:

- Time measure: \( \text{length} > \text{speed} \times \text{traject time step} \)
- Frequency measure: \( \text{length} > \text{speed} / \text{trajectory frequency} \)

Additional Motion Concepts

This section will consider additional tool motion details that are not covered in other sections, such as tool offsets in path following for a variety of tool motion and interpolation types, and how to handle TCFs that are attached to inner links rather than the last or terminal joint of the mechanism.

The CIMServer uses a dynamic trajectory generator, instead of pre-processing the motion segments; thus, the user must understand the importance of frame descriptions and how tool offsets are used to move relative to target frames or along paths which are defined as a set of motion segments.

Tool Motion Types

Presently, the tool TCF (tool or terminal control frame) is constrained to four tool motion types, as set by the API CxSetToolMotionType:

- CX_FULL_POSE
- CX_Z_POSE
- CX_Z_POSE_NO_SPIN
- CX_FIXED_ORIENT

and discussed in more detail in the following sections.

Any robot with two base branches of serially linked independent joints in the CODE hierarchical sense is classified as a CX_NC_ROBOT. Any robot with a single branch of serially connected independent joints is referred to as a CX_SERIAL_ROBOT.
**CX_FULL_POSE**

**CX_FULL_POSE** requires the mechanism to place the tool frame at the same position and orientation of a target frame unless an offset is specified. This type should not be used for robots or machine tools having less than 6 degrees-of-freedom, and consequentially, less than 6 joints. Interpolation between poses uses a screw vector and a translation to interpolate a tool frame from its initial orientation to its final target orientation. See Figure 4-6 below.

![Figure 4-6: CX_FULL_POSE Screw Vector](image)

**CX_FULL_POSE** should only be used when mechanisms have a full orientation capability. Examples of mechanisms that do not have this capability are 5-axes welding type robots that do not require the tool to spin about an approach vector, or machine tools that may have only one or two orientation axes, or, in the most common case, none at all, such as a face mill.

**CX_Z_POSE and CX_Z_POSE_NO_SPIN**

**CX_Z_POSE** assumes that the frames are arranged relative to the mechanism such that the mechanism tool Z axis can be aligned with a target Z axis. This pose type is useful when it is necessary to maintain a tool approach vector in some preferred normal direction relative to a surface or edge. Interpolation between two poses occurs in two steps, depending upon robot type: by 1) determining a vector normal to the TCF and target Z axes and then rotating about this vector from the initial TCF Z axis orientation to the target Z axis orientation, while simultaneously, and if the robot is a **CX_SERIAL_ROBOT**, 2) rotating about the tool Z axis to align the tool frame X-Y axes with the interpolated frame X-Y axes. Step two is ignored if the robot is **CX_NC_ROBOT** since the pose setting will actually default to the pose setting **CX_Z_POSE_NO_SPIN**.

**CX_Z_POSE** is useful for mechanisms that do not have a full orientation capability (typically they cannot spin the tool), but which can align an asymmetric tool with an axis or vector in space. Thus, **CX_Z_POSE** is treated differently for robots that are serial (**CX_SERIAL_ROBOT**) from robots such as machine tools that branch (**NC_ROBOT**). The **CX_SERIAL_ROBOT** is assumed to be the articulating type referred to as the modern industrial robot, whereas the **CX_NC_ROBOT** is assumed to have restricted orientation capability. By this we mean that the **CX_SERIAL_ROBOT** has the capability to spin the tool frame about its Z axis to align the tool X-Y axes with the target X-Y axes, whereas the **CX_NC_ROBOT** can align its Z axis, but not its X-Y axes.

**CX_Z_POSE_NO_SPIN** can be used to override spin about the tool Z axis. Although this remains the default **CX_Z_POSE** type setting for an **CX_NC_ROBOT**, it can also be used for **CX_SERIAL_ROBOTS**. For example, this would be the correct setting if the robot is used in sheet metal surface polishing or edge grinding where the grinding or polishing wheel is rotating at high rpm’s, but must be oriented in a preferred normal orientation relative to a surface or edge.
CX_FIXED_ORIENT

CX_FIXED_ORIENT assumes that the tool frame orientation is to be held constant during motion of the robot when the robot is engaged in curvilinear motion using a tool interpolation type of CX_LINEAR_INTERP, as specified by the API CxSetInterpType(). For the case of CX_JOINT_INTERP and a move to a target which uses inverse kinematics to determine the final joint values, the target frame is modified to have the same orientation as the initial tool frame.

CX_FIXED_ORIENT is the preferred tool motion type for gantry X-Y-Z robots and machine tools that have no orientation joints, particularly in target type moves that require inverse kinematics. By specifying CX_FIXED_ORIENT, the inverse kinematics routines effectively ignore the target frame orientation and choose joint values based on target position only. This releases the user from having to build a simulation model in which all target frames all aligned with the tool frame.

Tool Control Frame (TCF)

In CODE, the term frame is used to denote the position and orientation of a triad (set of XYZ axes) relative to another triad or frame. The tool control frame or TCF is used to specify the relative position and orientation of a robot tool frame relative to a target frame. Another commonly used term is TCP for tool or terminal control point, although this term does not incorporate the concept of tool orientation.

Tool Offsets

Tool offsets are used for tool standoffs from targets. For example, machining along a control surface such as a drive surface or a check surface requires a tool offset which can be set relative to a target frame. See Figure 4-7. For example, to move a cutter tool with diameter D along a drive surface, up to a check surface, the CODE API CxMoveRelNode would be specified with positional offsets X = D/2, Y = -D/2, and Z chosen to cut into the material the proper depth.

Similarly, robotic welding will often require a positional and orientational standoff. For example, see Figure 4-8, where the orientation standoff might be some $\alpha$ about the Y axis. If the algorithms seek to
maintain the tool orientation in some allowable welding cone, then algorithms may additionally specify a \( \beta \) rotation about the Z axis (not shown in Figure 4-8), as determined in customized kinematics routines.

![Figure 4-8: Orientation Standoff of a Tool](image)

**Inner TCFs**

In general, TCFs are attached to the terminal (distal) joint of a robot or machine tool and targets are attached to the world for **CX_SERIAL_ROBOT**'s and to the base branch for **CX_NC_ROBOT**'s. Occasionally, a TCF is attached to an inner joint of a **CX_SERIAL_ROBOT**. We refer to this case as the inner TCF. For example, a sensor may be too large to attach it to the last joint, such as a vision system used to acquire some feature in space.

For general target moves, the CIMServer automatically determines the parent joint of a TCF and sets the value of `inner_tcf` (member of the `command_entry` data structure) to the joint number of the parent. There is one problem here since the inverse kinematics routines are designed to match both position and orientation of the whole robot. An inner TCF may not be able to match the target orientation, but may be able to attain the target position, because a restricted set of joints are being used in the target move. And, in most cases, the outer joints of robots are designed for orientation. To handle inner TCF's the inverse kinematics routines have to be revised as described in Chapter 0 on CODE KINEMATICS.

The user should be aware that the reported orientation (screw) speeds may be in error if this tool motion type is specified, because the user modifies the joint values after the orientation speed calculations have been made.

**Motion Parameters**

Next, we consider important motion data structures and parameters which can be used in custom motion routines supplied by the user. The user can review these data structure parameters in the include files `server.h`, `robot.h`, and `joint.h`, listed under the directory “~ROBTOP/include/code” where ROBTOP refers to the directory under which CODE was installed.

**WARNING!** Although the user has access to the source code listings for `server.h` and other data structures, the user must be extremely cautious in using parameters in custom motion routines that are not described in the following sections.
mechanism_entry data structure
The mechanism_entry data structure accesses the motion queue (which functions as a rotary buffer) through several critical parameters.

```c
robot *rob; /* pointer to robot data structure */
command_entry q[CX_QUEUE_SIZE]; /* accesses moves in the queue */
long queued_command; /* next command in queue */
long current_command; /* current command in queue */
long last_move; /* last move in queue */
```

The robot and command_entry data structures are described in sections which follow. The queued_command, current_command, and last_move parameters are used to access the moves in the queue and to indicate how the queue is being updated, according to the following conditions:

- `queued_command != current_command`
  Implies command pending in queue (queued_command) which can either be loaded after current_command move completed (CX_MOVE_WAIT or CX_MOVE_TO) or blended with current_command move (if CX_MOVE_THRU), depending upon blending policy and move type.

- `current_command != last_move`
  Implies that blending underway since two moves are being processed simultaneously or that the current_command has completed itself during the blend but the last_move is still undergoing deceleration.

- `queued_command == current_command`
  No other move currently in queue to process; if so, the user should force a blending policy of CX_MOVE_TO to stop robot at end of move.

command_entry data structure
The command_entry data structure contains most of the move and status parameters. The most important ones follow:

- `long type; /* only CX_ROB_MOVE type to be used */`
- `long ramp_status; /* CX_RISE, CX_STEADY, or CX_FALL */`
- `long motion_status; /* current motion status along trajectory */`
- `long blend_policy; /* sets blending policy */`
- `long blend_status; /* to indicate if in blend or not */`
The ramp_status parameter can be used to sequence through the trajectory and to determine which acceleration values to use. Valid values to be used are CX_RISE (accelerate to increase speed), CX_STEADY (constant speed portion of the trajectory), or CX_FALL (decelerate to reduce speed).

The motion_status parameter is used to initialize and flag the motion state as one or two moves are being processed. Queued moves will be initialized by the CIMServer to CX_START_MOTION. This value can be used in custom routines to perform any necessary initialization, such as loading the tcf_initial and tcf_final frames for a Cartesian path move (discussed in a following section). Immediately after initialization, the user should set the status to CX_IN_MOTION. The user can set the following values (see robconst.h) for the motion_status of a move:

- **CX_IN_MOTION** Used to indicate that motion initialized and move now being processed
- **CX_END_OF_MOTION** Used to indicate that last trajectory step completes move
- **CX_NO_MOTION** Used to indicate that queued move target same as current TCF

The user_motion.c routine user_trajectory expects to return a motion status value directly from the trajectory generator, as described in more detail in the section user_trajectory. This status will be the same as the queued motion status for a single (non-blended) move (last_move == current_command), but should be set to one of the following values for a blended move where two queued moves (last_move!= current_command) are being processed. Note that the first status member (e.g., the IN in CX_IN_MOTION_NO) refers to last_move while the last status member (e.g., the NO in CX_IN_MOTION_NO) refers to current_command.

- **CX_IN_MOTION_IN** in the midst of blending last_move with current_command
- **CX_IN_MOTION_NO** current_command target same as last_move
- **CX_IN_MOTION_END** current_command completes while last_move does not
- **CX_END_MOTION_END** last_move and current_command simultaneously complete
- **CX_END_MOTION_IN** last_move completes first
- **CX_END_MOTION_NO** last_move completes and current_command target same as last_move

Two blending parameters can be used to govern the blending process: blend_policy and blend_status. The blend_policy is set by the API CxSetBlendPolicy and determines whether blending is to be used, and whether a move is to be completed before returning from a move API such as CxMoveRelNode. Typical values for blend_policy are as follows:
CX_MOVE_WAIT (Default) blending off and move completes before returning to calling routines

CX_MOVE_TO Blending off and moves are immediately loaded into move buffer and then return to move routine

CX_MOVE_THRU Blending of contiguous joint or Cartesian curvilinear moves

The blend_status parameter can be used to set and modify the blending status according to move circumstances. Valid values are:

CX_NOT_IN_BLEND All moves in queue initialized to this value

CX_IN_BLEND To indicate that blending underway

CX_DO_NOT_BLEND Used to override blending instructions

type = CX_ROB_MOVE

The user must check the queue type to determine if the queued command is an actual robot move. Given the mechanism_entry data structure and a pointer mech (mechanism_entry *mech), the user can examine the next member of the queue to determine if the queued member is a robot move:

if ( mech->q[mech->queued_command].type == CX_ROB_MOVE )
    " next queued member is a robot move "

Other motion related commands can be loaded into the queue, but should not be accessed by the user. Only CX_ROB_MOVE motion types should be accessed by the user.

WARNING!: The user must not use any queue members that do not have queue type equal to CX_ROBMOVE.

check_next_command(mechanism_entry *mech)

The routine which the user should use to increment to the next queued move is called check_next_command. Its only argument is the mechanism_entry data structure and it returns a long function value. This function can be used by the user to automatically increment to the next move command of the type CX_ROB_MOVE in the queue by simply calling it. If there is no move command pending in the queue, this function will return the function value CX_NO_COMMAND_PENDING; else, the function will return the function value CX_COMMAND_PENDING.

Since the buffer is rotary, the queue number will always have values beginning with zero and less than some maximum CX_QUEUE_SIZE.
Additional Parameters

The user should be aware of the following command_entry parameters, differentiated by those which can be changed or initialized in the trajectory routines, and those which are settings and should not be changed.

**Changed or initialized in trajectory routines:**

```c
long jnt_slow; /* jnt taking longest time for joint move */
double *dof; /* current joint values */
double *dof_act; /* current joint actuator values */
double *dof_initial; /* initial jnt values in a jnt move */
double *dofinal; /* final jnt values in a jnt move */
matrix tcf_initial; /* initial TCF pose frame for move */
matrix tcf_final; /* final TCF pose frame for move */
matrix tcf_target; /* interpolated target frame computed foreach trajectory step */

double total_dist; /* total move length */
double moving_dist; /* dist into seg (<= total_dist) */
double total_screw_ang; /* total screw angle (in radians) for CX_FULL_POSE */
double total_screw_ang_z; /* total screw angle (in radians) about z axis for CX_Z_POSE */
double *curspd; /* current joint rates */
double *curspd_act; /* current actuator joint rates */
double *curaccel; /* current joint accel rates */
double *curaccel_act; /* current actuator joint accel rates */
double *jnt_setspd; /* desired joint rates */

double tip_spd; /* current tool curvilinear speed, unit/s */
double screw_spd; /* current screw speed, rad/s */
double screw_spd_z; /* current screw speed about z axis, rad/s */
double ramp_time_rise; /* time set for speed increase */
double ramp_time_fall; /* time set for speed decrease */
double ramp_accel_rise; /* acceleration value for speed increase */
double ramp_accel_fall; /* deceleration value for speed decrease */

double screw_accel_rise; /* acceleration value for rotational speed increase */
double screw_accel_fall; /* deceleration value for rotational speed decrease */
long rotate_flg /* TRUE or Z_pose if rotation dominates distance */
```

**Settings not to be changed:**

```c
double jnt_set; /* fractional speed setting: 0 < jnt_set < 1.0 */
double tip_set; /* tool curvilinear speed setting, unit/s */
```
double screw_set; /* screw speed setting, rad/s */
long jacobian_flag; /* TRUE if jnt rate checkin on */
long jnt2act_map; /* joint to actuator class for robot */
long tool_motion_type; /* CX_FULL_POSE, or CX_FIXED_ORIENT, etc. */
long accel_flg; /* CX_CONST_RAMP_TIME or CX_CONST_RAMP_ACCEL */

The user can call the routine get_target_frames(q, this_error) to initialize tcf_final and tcf_initial, but should only call this routine for motion_status = CX_START_MOTION, unless motion tracking is on. In the CIMServer's trajectory generator, motion tracking is not used during CX_MOVE_THRU blending, but only for CX_MOVE_WAIT or CX_MOVE_TO policies.

For curvilinear moves the trajectory generator must determine, at each trajectory step, the desired tcf_target to be supplied to the inverse kinematics function pick_soln, which is called in user_trajectory()

In joint type moves the trajectory generator must return new joint values stored in the joint array dof for the incremented robot configuration. The robot data structure holds parameters which must be initialized at CX_START_MOTION.

robot data structure

The robot data structure holds parameters which are robot constants and additional parameters which vary. Useful parameters which can be accessed through the robot data structure are:

Changed or initialized in trajectory routines:

double *dofold; /* last joint values */
double *dofold_act; /* last joint actuator values */
double *old_speed; /* joint speeds at last configuration */

Settings not to be changed:

mechanism_entry *mech; /* pointer to mechanism_entry */
joint **joints; /* pointers to joint data structures */
long ndof; /* number of independent joints */
long njoint; /* number of all joints */
double accel_rise_max; /* max accel value for speed increase */
double accel_fall_max; /* max accel value for speed decrease */
double screw_accel_rise_max; /* max accel for screw spd increase */
double screw_accel_fall_max; /* max accel for screw spd decrease */
double ramp_time_rise_min; /* min time for speed increase */
double ramp_time_rise_min; /* min time for speed decrease */

dofold holds the joint values in the last configuration and is useful for approximating joint rates using simple equations like
q->curspd[i] = (q->dof[i] - rob->dofold[i])/(*traject_time);

dofinal and dofinitial are used to establish the limit values in a joint trajectory move and the command_entry member, jnt_slow, can be used to record, in a joint interpolated move, which joint governs the trajectory motion.

**joint data structure**

The joint data structure holds parameters which are robot constants and other parameters which change. The joint i data structure can be accessed by rob->joints[i]. Useful parameters which can be accessed through the joint data structure are:

**Changed or initialized in trajectory routines:**

```c
double accel_rise;  /* desired accel value for jnt speed increase */
double accel_fall;  /* desired accel value for jnt speed decrease */

double dofmin;      /* joint minimum limit, CAUTION */
double dofmax;      /* joint maximum limit, CAUTION */
```

**Settings not to be changed:**

```c
double jtype;        /* joint type (CX_TRANS or CX_ROTATE) */
double maxspd;       /* joint maximum speed */

double accel_rise_max; /* max accel for jnt speed increase */
double accel_fall_max; /* max accel for jnt speed decrease */
```

dofmin and dofmax are joint limits that may dynamically change for certain robots - see Chapter 1 in *Software Maintenance and Modification* on Robot Kinematics.

**WARNING!**: dofmin and dofmax may be changed by the user, if they are restricted to values less than the physical limitations of the robot, but the user must be cautious in doing so. If the user, in error, increases these beyond the physical limits or those limits programmed into the physical controller, damage to the robot or other devices in the cell may occur.
Motion Interfaces

CODE provides several interfaces that an expert user can use to program robot motion, plus some additional routines to facilitate initialization. These interfaces use four routines in the file user_motion.c which is provided to the user in source code form in the directory $ROBTOP/lib/cimetrix/custom where $ROBTOP is an environment variable defining the directory under which CODE was installed:

user_trajectory - routine which branches by tclass number (trajectory class) to a user defined function to calculate the joint changes which occur in a joint or Cartesian move. User defined functions are called in the include file user_motion.FG (FG stands for frame generator). The tclass number should be some unique integer not in the range 0 - 999 which is reserved for the default CIMServer algorithms. The user must examine user_motion.FG to determine a unique number.

user_joint_rates - routine which branches by jclass number (joint rate class) to either a CIMServer default or user specified routine for calculating joint rates in either joint or Cartesian moves. This number is set by default to 0 in the robot editor. A case switch allows the user to enter a user developed routine with the call added in the include file user_motion.JM. The number entered should be some unique integer not equal to 0 (reserved for CIMServer algorithms). The user must examine user_motion.JM to determine a unique number.

user_convert_act_to_jnt - routine which branches on jnt2act_map parameter, defined in the command_entry data structure, to a user defined routine that converts from actuator space to joint space. The call to the user specified routine should be included in the file user_motion.AJ. This interface routine is invoked before calling the CIMServer’s forward kinematics routine which work in joint space. The number entered should be some unique integer not equal to 0 (reserved for the CIMServer default which is a one-to-one mapping). The user must examine user_motion.AJ to determine an available unique number.

user_convert_jnt_to_act - routine which branches on jnt2act_map parameter, defined in the command_entry data structure, to a user defined routine that converts from joint space to actuator space. The call to the user specified routine should be included in the file user_motion.JA. This interface routine is used when connecting to an open architecture controller and is called before a move is sent to the servo card. The number entered should be some unique integer not equal to 0 (reserved for the CIMServer default which is a one-to-one mapping). The user must examine user_motion.JA to determine an available unique number.

The user should note that it will not be necessary to modify these routines since the user only needs to insert routines into include files which are then inserted into the main body of these interface routines at
compile time. Each of these routines allows the user to specify a unique classification number that is used by a case switch in user-modifiable include files to branch to the routine of user choice. These classification numbers are entered by the user in the robot editor during a CIMTools user session and, in most cases, must be uniquely defined for each robot.

**user_trajectory**

The function of user_trajectory is to return a feasible trajectory time, joint values, joint speeds, and possibly joint accelerations which fall within the capabilities of the robot, and at the incremented trajectory configuration. The user should open the file user_motion.c in the directory $ROBTOP/lib/cimetrix/custom and review the organization of the function.

user_trajectory(), which is similar to the organization depicted in Figure 4-5. The user may also wish to view the organization of the CIMServer's default trajectory interface routine, robline_frame_gen, which will not be discussed in this section.

Users can enter their own trajectory routines by adding call(s) to their specific routines in the include file user_motion.FG, which will be inserted into the routine usertrajectory at compile time. user_motion.c is available in source form for compiling and linking custom user routines. The call is inserted in the case switch for a unique tclass number specified either through the API set_tclass_number or in the CIMTools robot editor. The CIMServer's default frame generator, robline_frame_gen(), is called in user_trajectory().

Three arguments are passed through user_trajectory():

```c
mechanism_entry *mech;       /* mechanism data structure*/
double *traject_time;        /* desired trajectory increment time*/
command_entry **q;           /* address of pointer to current move command */
error_msg *this_error;       /* returns error message */
```

As described in the previous sections, the mechanism_entry data structure allows the user to access the queued move commands and other critical parameters through the mechanism_entry, command_entry, robot, and joint data structures.

q is passed in as a double pointer to allow the user to change this address locally (e.g., to point to another queue member). Since mech has access to the command_entry queue, the user can define local queue pointers which can be set to any queue member. Of course, the only members of the type CX_ROB_MOVE that should be accessed are the current_command, the last_move if it differs from current_command, and, possibly, the queued_command by using the routine check_next_command. The user need not worry about the error_msg argument, but simply return CX_ERROR if user_trajectory() returns CX_ERROR.

For curvilinear moves user_joint_rates() must be called from user_trajectory(). The user will notice, by referring to robconst.h, that all joint moves have integer numbers < CX_JOINT_MOVE, and all curvilinear type moves have integer numbers > CX_JOINT_MOVE.

**WARNING!:** The trajectory time (traject_time) can be increased by the user custom routines, but not decreased.
Initialization Routines

Three routines can be used to assist the organization of custom trajectory routines: valid_tcf_target, get_target_frames, followed by motion_start. These routines are to be called only for the case of motion_status set to CX_START_MOTION if motion tracking is off (the API CxSetMotionTrack is used to turn tracking on), but can be called as needed to update the target information if motion tracking is on. Tracking is only valid for CX_JOINT_INTERP and CX_LINEAR_INTERP and will not work for CX_CIRCULAR_INTERP or for paths defined as curvesegs.

It is assumed that the command_entry queue member is passed into the custom routine as **q. The queued member passed as argument into user_trajectory is always the current_command, which for most cases is the active move being processed in the trajectory routines. But there is one special case the user should check. If blending is being used, and current_command completes (for example, the deceleration distance of last_move is substantially longer than the move distance of current_command) before last_move completes, then it may be possible that there are no additional moves in the queue to continue the blend with last_move. The user can detect this situation by determining if current_command == queued_command and the motion status of current_command is CX_END_OF_MOTION.

NAME
valid_tcf_target - validate that target node exists and that tcf node is attached to robot.

C SYNTAX
long valid_tcf_target(command_entry *q, error_msg *this_error)

where
q - pointer to current command_entry data structure

RETURN VALUES
Function returns 0 if successful, else CX_ERROR returned.

NAME
get_target_frames - get dof_initial and dof_final for joint moves and tcf_initial and tcf_final for curvilinear moves

C SYNTAX
long get_target_frames(command_entry *q, error_msg *this_error)

where
q - command_entry data structure

RETURN VALUES
Function returns 0 if successful, else CX_ERROR returned.

NAME
motion_start - initialize the move parameters and set the motion status to CX_IN_MOTION.
C SYNTAX

long motion_start(command_entry *q, error_msg *this_error)

where

q - command_entry data structure

RETURN VALUES

Function returns 0 if successful, else CX_ERROR returned.

If valid_tcf_target() returns an CX_ERROR, then the target or tcf node id is not valid and the user’s custom routine should return an CX_ERROR likewise. The call to this routine should be:

    if(valid_tcf_target(*q, this_error) == CX_ERROR) return(CX_ERROR);

For curvilinear moves get_target_frames() can be called to determine the starting and ending frames (q->tcf_initial, q->tcf_final) for a move. It should be called before motion_start() is called. The call to this routine should be:

    if (get_target_frames(*q, this_error) == CX_ERROR) return(CX_ERROR);

For curvilinear moves the user would call get_target_frames() with the arguments (*q, this_error) to load tcf_initial and tcf_final into q. Note that tcf_initial is loaded inside the routine.

If the move type is a joint move (as specified by interp_type), get_target_frames() will determine the starting and ending joint values (q->dofinitial, q->dofinal), while motion_start() will determine which joint takes longest to accomplish the move, load that joint number into jnt_slow, and initialize the joint speed according to the setting specified by the API CxSetJointSpeed and the parameter jnt_set in the command_entry data structure.

If a curvilinear type move, motion_start() will determine the move segment information necessary for the trajectory step calculations and initialize the move segment by move type (CX_JOINT_INTERP, CX_LINEAR_INTERP, CX_CIRCULAR_INTERP).

Inside motion_start() a routine determines if screw motion dominates over linear motion. The flag set for this purpose is the parameter rotate_flg in the command_entry data structure which is set to CX_FALSE by default. If the rotate_flg is set to CX_TRUE or CX_Z_POSE, then screw motion dominates the linear motion. If CX_TRUE, this flag can then be used in custom routines to increment the rotation and proportion the linear portion of the move. If CX_FALSE, then the linear move is incremented and the rotation proportioned.

If the rotate_flg = CX_Z_POSE, the spin (screw_spd_z) about the Z axis dominates both the linear motion and the screw rotation. The call to this routine should be:

    if(motion_start(*q, this_error) == CX_ERROR) return(CX_ERROR);

Several parameters are set in motion_start() for both joint and curvilinear moves. These can be used to govern the steps along the trajectory:

    double total_dist; /* total move length (rad if rotary joint */
    double moving_dist; /* dist into move */
    double total_screw_ang; /* total screw angle (rad) */
    double total_screw_ang_z; /* total spin screw angle(rad) */
The following parameters are initialized to the following values:

```c
double moving_dist = 0.0; /* dist into move */
double screw_ang = 0.0; /* screw angle (rad) */
double screw_ang_z = 0.0; /* screw angle about z(rad) */
double tip_spd = 0.0; /* linear speed (unit/s) */
double screw_spd = 0.0; /* screw speed(rad/s) */
double screw_spd_z = 0.0; /* screw speed about z(rad/s) */
```

**user_joint_rates**

The function of `user_joint_rates` is to determine, for each step along the trajectory, joint rates and then determine if these rates exceed the joint limits. Several parameters are passed as arguments:

```c
command_entry *q; /* current_command move */
command_entry *q_prev; /* last move if blending */
long *exceed_jnt_rate; /* TRUE if exceed joint rates, else FALSE */
double *traject_time; /* trajectory time step */
error_msg *this_error; /* returns error message */
```

The user can enter a custom routine in the include file `user_motion.JM`, by specifying a unique `jclass` number. The function of this routine is to determine the joint rates of each joint at the new target configuration, and to respond to cases where the joint rates exceed their allowables. The parameter `exceed_jnt_rate` is provided to flag the case of joint rates exceeded. The user can enter any custom routines to handle excessive joint rates.

Once `user_joint_rates()` has been called from `user_trajectory()` and returned, the user should provide a new set of joint values (`dof`), a set of joint speeds (`curspd`), and an allowable `traject_time` that is equal to or greater than the value originally provided to `user_trajectory`. The pointer `*traject_time` is provided so that its value may be increased, if necessary.

**user_convert_jnt_to_act, user_convert_act_to_jnt**

If the user is driving a physical mechanism, it may be necessary to write a custom routine to convert from joint space (`dof, curspd`) to actuator space (`dof_act, curspd_act`). The servocard may issue commands in terms of encoder increments and possibly to secondary actuators such as ball screws or hydraulic actuators.

User functions to allow conversion of joint space to actuator space can be supplied in the include file `user_motion.JA` for a unique `jnt2act_map` number.

Note that this function is called before a move is sent to the servo sub_system. If the user enters a custom routine, there must be a corresponding conversion back to joint space by a custom function entered in the include file `user_motion.AJ` in `user_convert_act_to_jnt`.

The user can examine the include files for examples.

**Example: Custom User Trajectory**

The following example shows how a custom trajectory generator can be interfaced to the CIMServer.
Theory behind Example Trajectory Generator
The example included below is an S-curve trajectory generator. The name comes from the shape of the velocity profile that it generates, shown in Figure 4-9. An S-curve ramp, rather than a linear curve ramp, results in a smoother transition from zero velocity to constant velocity.

![Figure 4-9. S-Curve Velocity Profile.](image)

The method employed uses constant jerk to generate the S-curve. Jerk is the time derivative of acceleration and is the driving variable behind the trajectory generation. Through proper planning the jerk can be specified such that it results in the desired acceleration, velocity and distance at each point. Four equations which govern the relationship between these four variables follow (some are derived from others). These equations are derived by assuming constant jerk

\[
\begin{align*}
(1) \\
(2) \\
(3) \\
(4) \\
(5)
\end{align*}
\]

where
\[
\begin{align*}
x &= \text{distance} \\
v &= \text{velocity} \\
a &= \text{acceleration} \\
j &= \text{jerk} \\
t &= \text{time}
\end{align*}
\]

Note: (5) is only valid when initial acceleration and velocity are zero. Symmetry allows for its use along both the concave and convex portions of the ramps, however.

To generate the desired velocity profile in Figure 4-9 requires the following: the jerk be a constant, positive value during the concave portion of the ramp, the negative of that value during the convex portion of the ramp, and a zero value during the constant velocity portion of the profile. The ramp down
requires the same respective values of jerk for the convex and concave portions. The constant absolute value of jerk will be different for the ramp down if a different maximum acceleration is desired; however, the negative and positive natures for the convex and concave portions, respectively, remain the same.

The jerk profile, along with the resulting acceleration, velocity and distance profiles, is shown in Figure 4-10.

![Jerk, Acceleration, Velocity and Position Profiles](image)

**Figure 4-10. Jerk, Acceleration, Velocity and Position Profiles.**

**CODE Implementation**

Implementing a custom trajectory generator in the CIMServer involves modifying an existing CIMServer include file and creating a new function (usually contained in a new file). The new function call in this example is `custom_frame_gen`, and the file in which it resides is `custom_frame_gen.c`.

**Modification of CODE Include File user_motion.FG**

The CIMServer include file that must be modified is `user_motion.FG` ("FG" stands for frame generator) found in `$ROBTOP/lib/cimetrix/custom`. This function contains a case switch that allows the CODE user to choose either the standard CODE trajectory generator or a custom generator. The case number for the new custom generator must be greater than 999. The number chosen for this example is 1500. Through an API (`set_tclass_type`) or by using CIMTools, a CODE user can specify the number of the desired case. The modified include file is shown here.

```c
case 0: /* cases 0, 1, 2 are used in CIMServer trajectories */
case 1:
case 2:
    status = robline_frame_gen(mech, traject_time, q, this_error);
    return(status);
    break;

/*** case added for writing PVT data to a file ***/
```
case 999:
    status = robline_frame_gen(mech,traject_time,q,
        this_error);
    filePVT((*traject_time), (*q));
    return(status);
    break;

    /* user trajectory functions can be added as a unique case number >= 1000; NOTE that 0-999
    are reserved for the default frame generators developed by Cimetrix, Inc. To avoid problems with future
    CIMServer releases, please use numbers >= 1000! */

    case 1500:

#define CX_NOT_FINE_ENOUGH_RESOLUTION 3060

    if(((*q)->blend_policy == CX_MOVE_TO ||
        (*q)->blend_policy == CX_MOVE_WAIT) &&
        (*q)->interp_type == CX_LINEAR_INTERP &&
        (*q)->path_flg == CX_ROBOT_MOVE &&
        rob->jclass == 1500){
        status = custom_frame_gen(traject_time,q, this_error);
    }
    else{
        fprintf(stderr,"Switching back to standard trajectory
        generator and joint checker.\nNot all conditions for using the example generator were met.\n");
        rob->tclass = 0.0;
        rob->jclass = 0.0;
        status = robline_frame_gen(mech,traject_time,q,
            this_error);
        return(status);
    }
    if ( status == CX_NOT_FINE_ENOUGH_RESOLUTION ){
        fprintf(stderr,"Switching back to standard trajectory generator.\n");
        rob->tclass = 0.0;
        rob->jclass = 0.0;
        status = robline_frame_gen(mech,traject_time,q,
            this_error);
        return(status);
    }
    break;

As seen in the example above, two arguments are passed into the custom generator when
custom_frame_gen is called: (1) the trajectory time step,
*traject_time, and (2) a pointer to a structure, **q. These two arguments contain all the
information necessary for the custom generator to effectively interface with the CIMServer. The double
pointer q points to a structure, command_entry, that contains information on the current move. This
information includes, among other things, the current frame, the final frame to move to, the constant
velocity of the move, the maximum acceleration permitted, and the motion status. The custom frame
generator must then provide intermediate frames that correspond to the desired velocity profile and
finish at the given final frame.
Limitations of the Example Trajectory Generator

This example generator only covers the following conditions: (1) the blend policy is CX_MOVE_TO or CX_MOVE_WAIT (this means the mechanism will come to a complete stop at the end of each move), (2) the interpolation must be linear, (3) the move must be between two frames in the workspace as opposed to along a given path, (4) the example joint rate checker is also being used (see next section), and (5) the following expression is satisfied:

\[ \text{(7)} \]

The expression determines whether there are six points on an S-curve ramp. If it fails there are not. This check is made, however, within a sub-function that returns a message (CX_RESOLUTION_NOT_FINE_ENOUGH) if the expression is not satisfied.

If these conditions are not met, the program defaults to the standard CIMServer frame generator, which is equipped to handle other cases.

Addition of Function for Trajectory Generation

As mentioned above, the name chosen for this custom generator is custom_frame_gen. The function resides in $ROBTOP/ccode/ROBLINE/custom_frame_gen.c. Parts of the code are inserted in this section for discussion; the file in its entirety is listed at the end of this section.

Initialization

At the beginning of the function a check is made to see if the move is just beginning; if so, then the routine custom_motion_init is called:

```c
if((*q)->motion_status == CX_START_MOTION){
    if(custom_motion_init(&traj,q,traject_time,this_error) ==
       CX_NOT_FINE_ENOUGH_RESOLUTION)
       return(CX_NOT_FINE_ENOUGH_RESOLUTION);
}
```

Inside this routine, several important CIMServer public functions are called:

```c
if(valid_tcf_target(*q, this_error) == CX_ERROR) return(CX_ERROR);
if(get_target_frames(*q,this_error) == CX_ERROR) return(CX_ERROR);
if(motion_start(*q,this_error) == CX_ERROR) return(CX_ERROR);
if((*q)->motion_status == CX_NO_MOTION) return(CX_NO_MOTION);
```

The first call checks to see that the tcf and target nodes are valid. The second call, get_target_frames(), finds the initial and final tcf frames. The third call calculates the total change in position between the first and last frames and stores it in (*q)->total_dist. It also calculates the total change in orientation and stores it in (*q)->total_screw_ang. Finally, it checks whether linear or rotational movement dominates the move; the more dominant one will then...
drive the trajectory generation calculations and the other be scaled appropriately. The motion status is now set to CX_IN_MOTION.

It is recommended that these three calls be made in any custom trajectory generator.

The following segment uses the flag (*q)->rotate_flg, set in motion_start() to establish linear or rotational dominance, and to decide whether the local trajectory variables vel_set, accel_set and total_dist are driven by linear or rotational variables.

```c
if((*q)->rotate_flg == CX_TRUE){
    traj->vel_set = (*q)->screw_set;
    traj->accel_set_rise = (*q)->screw_accel_rise;
    traj->accel_set_fall = (*q)->screw_accel_fall;
    traj->total_dist = (*q)->total_screw_ang;
}
else{
    traj->vel_set = (*q)->tip_set;
    traj->accel_set_rise = (*q)->accel_rise;
    traj->accel_set_fall = (*q)->accel_fall;
    traj->total_dist = (*q)->total_dist;
}
```

The next segment of the initialization routine checks to see if there is enough distance to reach the constant velocity portion of the profile. If not, it lowers the target velocity so that it has just enough distance to ramp up and then ramp down. The routine then checks to see if there are enough points on the ramp to plan the ramp well (There must be at least six points total).

The last segment of the initialization routine sets initial values of acceleration, velocity, and distance to zero, and calculates the absolute value for jerk from (5).

### Trajectory Generation

Having made the call to the initialization routine, the function now makes a trajectory calculation by calling calc_jerk() and calc_traj(). These routines are called continually as long as the variable (*q)->motion_status remains CX_IN_MOTION. The routine works for either linear or rotational trajectory generation; as mentioned above, it is used for the more dominant one and the other is scaled appropriately at the end of custom_frame_gen().

### Determining Location in the Profile and Calculating Jerk

The program must periodically change the value of jerk as it moves through the profile. It must change the sign of the value in order to transition from concave to convex or vice versa. It must set the value to zero to create the steady portion of the profile.

The program begins in the CX_RISE_CONCAVE portion of the profile; in the initialization routine the flag traj->profile-status is set to CX_RISE_CONCAVE and the jerk is set to jerk_const_rise. In order to know when it should make the succeeding changes in the jerk value (and to adjust the trajectory time step, if needed), the program makes certain checks to see where it is in the profile. This is done in calc_jerk().

The program uses nested if-else blocks to make these checks. It checks first for the rise-concave portion and if the check is not satisfied it falls into the else block. It then checks for the rise-convex portion; if
this is not satisfied, it falls into the next else block, and so on. It continues until it finds the correct portion of the profile, setting the appropriate jerk value when it does.

**Adjusted Time Step** - The program makes each check two trajectory time steps ahead, because it is unlikely that an integer number of time steps will fit into each portion of the profile. The program must adjust the time step temporarily to catch the exact point at which to transition to the next portion of the profile. Otherwise, the planning for the whole trajectory can be thrown off.

CODE stipulates that the time step can only be made larger, not smaller. By checking two steps ahead the program can calculate the fraction of the time step right before the transition to the new portion of the profile, add it to the immediately-preceding time step, and present the sum to CODE as the adjusted time step. The fraction of the time step is calculated from (4). The first time the program fails the test, the current portion of the profile, it does not immediately begin the next portion of the profile. If first calculates an adjusted time step that brings it to the end of the current portion. The following iteration resets the time step and begins the next portion of the profile.

**Rise-Concave** - The first profile test checks to see if the current velocity is less than half the desired constant velocity and that the profile_status is CX_RISE_CONCAVE (defined constants are listed at the top of the custom file). If so, the program is in the concave portion of the rising ramp. If the profile_status is not CX_RISE_CONCAVE, it must be CX_FALL_CONVEX (see Figure 4-9). This means that the program is actually in the CX_FALL_CONCAVE portion of the profile.

**Rise-Convex** - If the velocity check fails, then the program falls into the succeeding else block. This usually indicates that it is now in the CX_RISE_CONVEX portion of the profile. The exception is the first time in, when it is actually finishing the CX_RISE_CONCAVE portion with an adjusted trajectory time step.

**Steady** - The check for the point where the profile should change from CX_RISE_CONVEX to CX_STEADY (or CX_FALL_CONVEX if there is not enough distance for a CX_STEADY portion) is made by watching the acceleration two steps ahead. The check is made to see where the acceleration changes sign. If the program did not set the jerk to zero at the appropriate time, then the acceleration would change sign and a new concave portion would begin. After recognizing that in two time steps the acceleration would change sign, the program uses (4) to calculate a time step that allows the profile to proceed just to the point where the acceleration reaches zero. At that point the jerk is set to zero, keeping the acceleration at zero along the entire STEADY portion of the profile.

**Fall-Convex** - The test for the transition from CX_STEADY to CX_FALL_CONVEX is done with distance, because the transition must occur when there is just enough distance left to decelerate. If there is too little distance to have a CX_STEADY portion, then the program automatically skips it.

While in the CX_STEADY portion of the velocity profile, the program checks two time steps ahead to see that there is enough distance to decelerate. If the test fails, the program calculates an adjusted time step to bring it to the end of the CX_STEADY portion. In the following iteration, the program resets the time step and begins the CX_FALL_CONVEX portion of the profile. If there is not enough distance for a CX_STEADY portion, the distance check will immediately fail and the program will transition directly from CX_RISE_CONVEX to CX_FALL_CONCAVE. It will first calculate an adjusted time step to bring it to the end of the CX_RISE_CONCAVE portion and then begin the CX_FALL_CONVEX portion in the next iteration with a reset time step.

**Fall-Concave** - The next check determines if the velocity two steps ahead falls below half the target velocity; if so, the program calculates an adjusted time step to finish off the fall-convex portion, finishes it, and falls into the CX_FALL_CONCAVE portion of the profile. The program should not fall into the
rise-convex portion of the profile (which partly uses the same check) because the preceding step was not rise-convex.

**Move Completion** - The final check, to see when the move should be completed, is done by checking acceleration. When it is determined that the acceleration will change sign (and all the previous checks have failed), then a final, adjusted time step is calculated that will finish the profile at the right point (see (4)). The flag (*q)->motion_status is set to CX_END_OF_MOTION to signal the CIMServer that the distance in the move has been covered.

After the trajectory time step is calculated (or left at the default value) and the value for jerk is determined (both are determined in the routine calc_jerk), the routine calc_traj calculates the resulting distance, velocity and acceleration from the equations of motion for constant jerk (see (1)-(3)).

**Calculation of Intermediate Frame from Trajectory Generation**

The last part of custom_frame_gen() uses the linear or rotational dominant flag to set the appropriate global CIMServer variables equal to the recently-calculated trajectory variables in preparation for returning to the CIMServer. If linear motion is dominant, then rotational motion is scaled from it and vice versa.

Finally, the program calculates the target frame resulting from the calculated distance and change in orientation. This is done in calc_targ(). The variable lin_vec is a unit vector in the direction of the change in position. The new position portion of a transformation matrix is obtained by multiplying the calculated target distance by this unit vector. Calling the CODE function rotm allows the change in orientation to be calculated from (*q)->screw_vec and (*q)->screw_ang, giving the resulting orientation portion of the same transformation matrix. Now, the initial frame can be multiplied by the transformation matrix to obtain the intermediate target frame corresponding to the current point in the trajectory profile.

The routine then returns the motion status to the CIMServer.

Note: The example custom trajectory generator must be used in conjunction with the example custom joint rate routine to follow. The CIMServer’s joint rate routine, when joint rate limits are reached, makes adjustments to the trajectory step by slowing down the motion. The example generator, however, is not equipped to handle this case. The example custom joint rate checker, in contrast to the CIMServer’s joint rate routine, simply prints joint rate violations to the screen and returns an error.

**Compilation**

The new function must be compiled with the CIMServer to take effect. The name of the file must be included in the makefile.

All the code for the example trajectory generator (except the include file) follows:

```c
/* Copyright (C) Cimetrix Inc. */

FILE

custom_frame_gen.c - This module provides routines for S-curve trajectory generation that can be used with the CIMServer. It was originally intended for an example and can handle only the simulation environment at this point. It is also restricted to the move-to, linear-interpolated, robot-move case, and can only be used in conjunction with the example joint rate checker found in custom_joint_rates.c.

PUBLIC FUNCTIONS
```
custom_frame_gen - interface routine for calculating S-curve trajectory
custom_motion_init - called at beginning of move to initialize parameters
calc_jerk - determines point along velocity profile and sets jerk accordingly; also adjusts trajectory
time step if necessary.
calc_traj - calculates distance, velocity and acceleration from given jerk and trajectory time step
values.
calc_targ - determines intermediate target frame from the new trajectory information.

/******************** IMPORTS **********************/
#include <code/roberro.h>
#include <code/robtypes.h>
#include <code/matrix.h>
#include <code/motion.h>

/******************** FORWARD DECLARATIONS **********************/
#define CX_RISE_CONCAVE 3050
#define CX_RISE_CONVEX 3051
#define CX_FALL_CONVEX 3052
#define CX_FALL_CONCAVE 3053
#define CX_NOT_FINE_ENOUGH_RESOLUTION 3060

typedef struct
{
    double dist,vel,accel,jerk,vel_set;
    double accel_set_rise,accel_set_fall;
    double total_dist,jerk_const_rise,jerk_const_fall;
    double dist_S_rise,dist_S_fall;
    long profile_status;
} trajectory;
static trajectory traj;

long custom_motion_init();
void calc_jerk();
void calc_traj();
void calc_targ();

/******************** PUBLIC FUNCTIONS *********************/

/--------------------------custom_frame_gen----------------------*

REMARKS
provides a custom motion interface to the dynamic mechanisms in the cell

FUNCTIONS CALLED
custom_motion_init
calc_jerk
calc_traj
calc_targ
-----------------------------------------------*
long custom_frame_gen(double *traject_time,command_entry **q,
error_msg *this_error)
{

/* If beginning of move, need to determine if new final frame
is valid target frame, if linear or rotational motion is
dominant, and what the total distance to be covered is. Also
need to initialize acceleration, velocity, and distance to
zero, and to initialize jerk to a constant value. */
if((*q)->motion_status == CX_START_MOTION){
    if(custom_motion_init(&traj,q,traject_time,this_error) ==
        CX_NOT_FINE_ENOUGH_RESOLUTION)
        return(CX_NOT_FINE_ENOUGH_RESOLUTION);
}

/* Check whether initialization call determined that move was
CX_NO_MOTION. If so, skip trajectory calculations. */
if((*q)->motion_status == CX_NO_MOTION)return(CX_NO_MOTION);

/* Begin trajectory calculations. First call calculates jerk
and makes a trajectory time step adjustment if necessary.
Second call calculates resulting acceleration, velocity and
distance; distance is what is needed to determine new
intermediate frame. Acceleration and velocity are needed for
succeeding calculations. */
calc_jerk(&traj,traject_time,q);
calc_traj(&traj,*traject_time);

/* End trajectory calculations. */
/* Determine intermediate target frame and speeds. Return
motion status. */
calc_targ(&traj,q);
return((*q)->motion_status);
}

/*************** PRIVATE FUNCTIONS ******************/

/*--------------custom_motion_init------------------
REMARKS
-determines validity of final frame, finds distance between initial and final frames, determines
whether linear or rotational motion dominates and sets local trajectory variables accordingly,
determines whether resolution allows for S-curve planning, determines whether distance allows for
target velocity and initializes motion parameters
FUNCTIONS CALLED
valid_tcf_target
get_target_frames*/
motion_start

void custom_motion_init(trajectory *traj, command_entry **q, double *traject_time, error_msg *this_error)
{

    /* Check to see that new final frame is a valid frame. */
    if(valid_tcf_target(*q, this_error) == CX_ERROR)
        return(CX_ERROR);

    /* Get tcf_initial and tcf_final frames */
    if(get_target_frames(*q, this_error) == CX_ERROR)
        return(CX_ERROR);

    /* Find total_distance and total screw angle between initial and final frames. These are stored in (*q)->total_dist and (*q)->total_screw_ang, respectively. This call also determines whether linear or rotational motion dominates the move. The flag (*q)->rotate_flg is set appropriately. */
    if(motion_start(*q, this_error) == CX_ERROR) return(CX_ERROR);

    /* Check for no-motion condition. */
    if((*q)->motion_status == CX_NO_MOTION) return(CX_NO_MOTION);

    /* Set local variables appropriately depending on whether move is linearly or rotationally dominant. */
    if((*q)->rotate_flg == CX_TRUE){
        traj->vel_set = (*q)->screw_set;
        traj->accel_set_rise = (*q)->screw_accel_rise;
        traj->accel_set_fall = (*q)->screw_accel_fall;
        traj->total_dist = (*q)->total_screw_ang;
    } else{
        traj->vel_set = (*q)->tip_set;
        traj->accel_set_rise = (*q)->accel_rise;
        traj->accel_set_fall = (*q)->accel_fall;
        traj->total_dist = (*q)->total_dist;
    }

    /* Calculate distances required to rise and fall */
    traj->dist_S_rise = traj->vel_set*traj->vel_set/
    traj->accel_set_rise;
    traj->dist_S_fall = traj->vel_set*traj->vel_set/
    traj->accel_set_fall;

    /* Determine if there is enough distance to rise and fall. If not, calculate new target velocity that can be obtained within distance. */
}
if(traj->total_dist < (traj->dist_S_rise + traj->dist_S_fall))
{
    traj->vel_set = sqrt((traj->total_dist * traj->accel_set_rise * traj->accel_set_fall)/(traj->accel_set_rise + traj->accel_set_fall));

    traj->dist_S_rise = traj->vel_set*traj->vel_set/traj->accel_set_rise;
    traj->dist_S_fall = traj->vel_set*traj->vel_set/traj->accel_set_fall;

}

/* Check to see if there is enough resolution to plan S-curve ramp. Must be at least six points on ramp. */
if((traj->vel_set/traj->accel_set_rise/(*traject_time) < 3.0 ) || (traj->vel_set/traj->accel_set_fall/(*traject_time) < 3.0))
{
    fprintf(stderr,"Not fine enough resolution to plan S-curve ramp./n");
    return(CX_NOT_FINE_ENOUGH_RESOLUTION);
}

/* Initialize accel, vel, and distance. */
traj->accel=0;
traj->vel=0;
traj->dist=0;

/* Calculate constant values of jerk for rise and fall. */
traj->jerk_const_rise = traj->accel_set_rise*traj->accel_set_rise/
traj->vel_set;
traj->jerk_const_fall = traj->accel_set_fall*traj->accel_set_fall/
traj->vel_set;

/* Initialize jerk to constant value of rise, and profile status. */
traj->jerk = traj->jerk_const_rise;
traj->profile_status = CX_RISE_CONCAVE;

return(OK);
}

/*------------------calc_jerk-----------------------

REMARKS
determines which stage of velocity profile the program is in and sets jerk and time_step (if necessary) accordingly.

FUNCTIONS CALLED

----------------------------------------------------*/

#include "trajectory_data.h"

/*--------calc_jerk--------*/

void calc_jerk(trajectory *traj,double *traject_time,
command_entry **q)
{
double vel_2stps_ahead;
double accel_2stps_ahead;
static double traject_temp;

/* Calculate what acceleration and velocity will be two time steps ahead. */
vel_2stps_ahead = traj->jerk*2.0*(traject_time) * (traject_time) +
traj->accel*2.0*(traject_time) + traj->vel;
accel_2stps_ahead = traj->jerk*2.0*(traject_time) + traj->accel;

/* Check to see if velocity will be less than half target constant velocity. If so, will still be in
CX_RISE_CONCAVE portion of profile (unless in CX_FALL_CONCAVE portion; for
this reason profile_status is checked for previous status). Set jerk appropriately. */
if( vel_2stps_ahead < 0.5*traj->vel_set &&
traj->profile_status == CX_RISE_CONCAVE ){
    traj->jerk = traj->jerk_const_rise;
    traj->profile_status = CX_RISE_CONCAVE;
}

/* If not, check whether accel will be changing sign. If not, then either finishing CX_RISE_CONVEX or in RISE_CONCAVE portion
of profile. If finishing CX_RISE_CONVEX, need to adjust trajectory step appropriately, and keep jerk at positive value. If in
CX_RISE_CONCAVE, need to set and keep jerk at constant negative value. */
else{
    if( accel_2stps_ahead > 0.0 && (traj->profile_status ==
CX_RISE_CONCAVE || traj->profile_status == CX_RISE_CONVEX)){
        if(traj->profile_status == CX_RISE_CONCAVE){
            traject_temp = *traject_time;
            *traject_time = 2.0*(0.5*traj->vel_set - traj->vel)/
(traj->accel_set_rise + traj->accel);
        }
        else{
            *traject_time = traject_temp;
            traj->jerk = -traj->jerk_const_rise;
        }
        traj->profile_status = CX_RISE_CONVEX;
    }
/* If the above two checks fail, check whether the distance
remaining is enough for deceleration. If so, then in CX_STEADY
portion or finishing CX_RISE_CONVEX portion. If finishing
CX_RISE_CONVEX, adjust time step. If not, set jerk and
acceleration to zero. Acceleration should already be zero,
however. */
else{
  if(traj->dist < (traj->total_dist - traj->dist_S_fall - 2.0)
   *traj->vel_set * (*traject_time)) {  
    if(traj->profile_status == CX_RISE_CONVEX) {  
      traject_temp = *traject_time;  
      *traject_time = 2.0*(traj->vel_set - traj->vel)
      traj->accel;
    }
    else {  
      *traject_time = traject_temp;  
      traj->jerk = 0.0;  
      traj->accel = 0.0;  
    }
    traj->profile_status = CX_STEADY;
  }
  /* If above three checks fail, check whether velocity has declined  
   to half its constant value. If not, then in CX_FALL_CONVEX  
   portion or finishing CX_STEADY or CX_RISE_CONVEX  
   portion of velocity profile (CX_STEADY may have been skipped  
   altogether). If finishing CX_STEADY or CX_RISE_CONVEX, adjust  
   time step and maintain current value of jerk. If in  
   CX_FALL_CONVEX portion, set and keep jerk at negative constant  
   value. */
  else{  
    if( vel_2stps_ahead > 0.5*traj->vel_set ) {  
      if(traj->profile_status == CX_STEADY) {  
        traject_temp = *traject_time;  
        *traject_time = (traj->total_dist - traj->dist -
         traj->dist_S_fall)/traj->vel_set;  
      }
      else {  
        if(traj->profile_status == CX_RISE_CONVEX) {  
          traject_temp = *traject_time;  
          *traject_time = 2.0*(traj->vel_set -
           traj->vel)/
           traj->accel;
        }
        else {  
          *traject_time = traject_temp;  
          traj->jerk = -traj->jerk_const_fall;
        }
      }
      traj->profile_status = CX_FALL_CONVEX;
    }
    /* If above four checks fail, check whether acceleration  
     will change sign. If not, then in CX_FALL_CONCAVE portion  
     or finishing CX_FALL_CONVEX. If finishing  
     CX_FALL_CONVEX, adjust time step and keep jerk at current value. If in  
     CX_FALL_CONCAVE, set and keep jerk at positive constant  
     value. */
  else{

if( accel_2stps_ahead < 0.0 ){
    if(traj->profile_status == CX_FALL_CONVEX){
        traject_temp = *traject_time;
        *traject_time = 2.0*(0.5*traj->vel_set - traj->vel)/
        (-traj->accel_set_fall + traj->accel);
    }
    else{
        *traject_time = traject_temp;
        traj->jerk = traj->jerk_const_fall;
    }
    traj->profile_status = CX_FALL_CONCAVE;
}
else{
    /* If all above checks fail, then ready to finish move. Adjust time step such that acceleration,
    velocity and distance will all be zero. Keep jerk at current value. Set flag for CX_END_OF_MOTION. */
    *traject_time = -2.0*traj->vel/traj->accel;
    (*q)->motion_status = CX_END_OF_MOTION;
    if((traj->dist - traj->total_dist)<CX_EPSILON)
        traj->dist = traj->total_dist;
    }
}

/*-------------------calc_traj-----------------------
REMARKS
given a value of jerk and a time step, it calculates distance, velocity and acceleration from the
equations of motion for constant jerk.

FUNCTIONS CALLED
----------------------------------------------------*/

void calc_traj(struct trajectory *traj,double *traject_time)
{
    /* Calculate distance, velocity and acceleration from the
equations of motion assuming constant jerk. */

    traj->dist += traj->jerk*(*traject_time)*(*traject_time)*
    (*traject_time)/6.0 + traj->accel*(*traject_time)*
    (*traject_time)/2.0 + traj->vel*(*traject_time);

    traj->vel += traj->jerk*(*traject_time)*(*traject_time)/2.0 +
    traj->accel*(*traject_time);

    traj->accel += traj->jerk*(*traject_time);
}
/*-------------------calc_targ-----------------------*/
REMARKS
calculates the new intermediate target frame from the dominant movement (linear or rotational)
calculated in trajectory and the scaled non_dominant movement.

FUNCTIONS CALLED
---------------------------------------------------*/
void calc_targ(trajecory *traj, command_entry **q)
{
    long i;
    vector vec;
    matrix rel_target;

    /* Check whether linear or rotational motion is dominant. If linear
    is dominant, scale angular movement accordingly. If rotational is
dominant scale linear movement accordingly. */
    if((*q)->rotate_flg == CX_TRUE){
        (*q)->screw_ang = traj->dist;
        (*q)->moving_dist = traj->dist*(*q)->total_dist/
        (*q)->total_screw_ang;
        (*q)->tip_spd = (*q)->screw_spd*(*q)->total_dist/
        (*q)->total_screw_ang;
    }
    else{
        (*q)->moving_dist = traj->dist;
        (*q)->screw_ang = traj->dist(*q)->total_screw_ang/
        (*q)->total_dist;
        (*q)->tip_spd = traj->vel;
        (*q)->screw_spd = (*q)->total_screw_ang(*q)->tip_spd/
        (*q)->total_dist;
    }

    /* “rel_target” is used to store move relative to tcf_initial */
    /* Multiply linear distance by unit vector to get relative position
    vector */
    for(i=0;i<3;i++)
        vec[i] = (*q)->moving_dist(*q)->lin_vec[i];

    /* Rotate matrix around “screw_vec” by amount “screw_angle” */
    rotm((q)->screw_ang,(q)->screw_vec,vec,rel_target);

    /* Multiply initial frame by transformation matrix “rel_target” to
    get final frame. */
    mul4x4((q)->tcf_initial,rel_target,(q)->tcf_target);

    /* in ref frame */
}

Example: Custom User Joint Rate Routine

The following example shows how a custom joint rate routine can be interfaced to the CIMServer.

Theory Behind Joint Rate Routine

When a move is made using linear interpolation, the CIMServer, or a custom trajectory generator, if used, plans the trajectory such that the speed and acceleration of the end effector (tip speed and tip acceleration) do not exceed those set by the user. However, this does not mean that the joints themselves, in order to satisfy the linear movement, will not be asked to exceed their rate limits. Even if no adjustments are made, the individual joint rates maybe calculated. The CIMServer has a default joint rate routine that calculates joint rates from changing joint angles and adjusts the trajectory move so as not to exceed those rates. The default routine uses a first-order difference equation.

The user may wish to use a custom routine to determine joint rates and to make any desired adjustments. This example shows how such a routine can be interfaced to the CIMServer. It uses a second-order difference equation, (7), to calculate joint rates from joint angles and the previous joint rates. The derivation of (7) follows:

\[
\text{(standard equation of motion)}
\]

\[
(7)
\]

where \( \theta = \text{joint value (translational or rotational (rad) joint)} \),

\[ = \text{joint rate (in rad/s if rotational joint)} \]

\[ = \text{joint acceleration (in rad/s}^2 \text{ if rotational joint)} \]

\( t = \text{trajectory step (seconds)} \)

and where \( \theta_0 \) and \( \dot{\theta}_0 \) are the joint value and joint rate, respectively, at the previous trajectory step.

For simplification purposes, the custom routine for computing joint rates does not make trajectory adjustments. It only returns an error if a joint rate limit is exceeded.
**CODE Implementation**

To implement the custom joint rate routine requires modifying an include file (`user_motion.JM`), and adding a function. The name of the added function will be `custom_joint_rates`.

**Modification of Include File `user_motion.JM`**

The modified include file, `user_motion.JM`, is included here:

```c
case 0:
    get_joint_rates(q,exceed_int_rate,(*traject_time),this_error);
    if(*exceed_int_rate && process_exceed_int_limit
        (q,q_prev,traject_time) == CX_ERROR) return (CX_ERROR);
    break;

case 1:
    fb_joint_rates(q,traject_time);
    break;

case 1500:
    if((q->blend_policy == CX_MOVE_TO || q->blend_policy == CX_MOVE_WAIT) &&
        q->interp_type == CX_LINEAR_INTERP &&
        q->path_flg == CX_ROBOT_MOVE &&
        q->tool_motion_type == CX_FULL_POSE)
    {
        custom_joint_rates(q,*traject_time);
    }
    else{
        get_joint_rates(q,exceed_int_rate,(*traject_time),this_error);
        if(*exceed_int_rate && process_exceed_int_limit
            (q,q_prev,traject_time) == CX_ERROR) return (CX_ERROR);
    }
    break;
```

Case 0 is the default CIMServer joint rate routine. Case 1 is another CIMServer joint rate routine developed for use with a force-ball device in a tele-operation environment. Case 1500 calls the example joint rate routine, `custom_joint_rates`. As can be seen in the code, the example routine only works for the case constituting the following: (1) the blend policy is `CX_MOVE_TO` or `CX_MOVE_WAIT` (each move comes to a complete stop before the next move begins), (2) the move is linearly interpolated, (3) the robot moves between two frames in space as opposed to along a path, and (4) the move is `CX_FULL_POSE`.

If these conditions are not satisfied, then the code defaults to the CIMServer’s joint rate routine. To choose a joint rate routine, a CODE user can either set the case number in an API (`CxSetJclassType`) in a CODE application process, or by using CIMTools.

**Addition of Function for Custom Joint Rate Calculation**

The file containing `custom_joint_rates()`, `custom_joint_rates.c`, is listed here:
void custom_joint_rates(command_entry *q,double traject_time)
{
    long i;/* joint loop counter */
    long ndof; /* number of joints */
    double deljnt;
    robot *rob;/* robot pointer */

    rob = q->rob;
    ndof = rob->ndof;

    for(i=0; i<ndof; i++){
        deljnt = q->dof[i] - rob->dofold[i];
        q->curspd[i] = 2.0*deljnt/traject_time - rob->old_speed[i];
        if(q->curspd[i] > rob->joints[i]->maxspd){
            fprintf(stderr,"Exceeded joint rate limits on joint \\
            %d\n",i);
            return;
        }
    }

    for(i=0; i<ndof; i++)
        rob->old_speed[i] = q->curspd[i];

    Both rob and ndof are used as local variables to save redirection time.
    The next step in the process is to find the change in joint angles for each joint (deljnt), and to apply (7) to calculate the current speed for each joint. If the speed is greater than the maximum for the joint then the function returns an error and an error message is printed to the screen. The last step of the code stores the current value of the speeds so they can be used for the calculations in the next iteration.

Compilation
The modified include file and the new function both need to be compiled and linked with the CIMServer executable. To make sure that the modified include file, and the new joint rate routine file, are linked in the code, the file user_motion.c must be recompiled. The makefile must also be modified to include the file custom_joint_rates.c.
CHAPTER 6

VecTool Interfaces

Introduction

The Vector Tool (VecTool) capability provides dynamic sensor modification of trajectory paths. VecTool will modify a tool path to move along or relative to some part feature, in response to sensor offsets which are generated dynamically by comparing the relative motion to some prescribed feature such as an edge or seam.

Two methods have been implemented: 1) model, and 2) dynamic. In the model-based method the tool is commanded to move to a known target along some prescribed path (line, arc, or a linked list of path segments called a curveseg). Offsets can be applied at each trajectory step as generated by the sensor.

The offset mode that can be applied for this method is either absolute or incremental. In the absolute mode the user will apply a matrix offset to the tool frame at each trajectory step. If the offset matrix is not modified, the previous offset will continue to be applied until it is changed or the motion is halted. In the incremental mode an offset deviation is applied to the absolute offset matrix as a path perturbation and the absolute offset matrix is updated by this deviation.

The CODE implementation provides the user a custom software interface to modify critical data structure members that are used in the trajectory generator. This provides the user tremendous flexibility in how and when the offsets are applied to the trajectory path. Since the custom routine runs in-line with the CODE trajectory generator, the response to sensor modification is almost immediate.

When VecTool is enabled the tool absolute and incremental offsets currently stored are set to zero. At each trajectory step, the user provides an updated offset which can be applied absolutely or incrementally. If no offset is generated by the user, the current offset will be applied to the model path. The motion will complete once the model endpoint target has been reached or mechanism motion is stopped. VecTool should not be turned off during this motion.

In the dynamic mode the tool is commanded to move relative to itself by an offset generated by the sensor. In a sense the tool essentially chases the sensor since it is generating tool offsets continually. A client process will issue a CxMoveRelTool to initiate the dynamic motion. As long as the sensor continues to generate offsets (which can be applied absolutely or incrementally) that modify the ending pose of the tool, the motion will continue. This motion is much like CODE’s motion tracking capability. The motion only completes when VecTool is turned off or the mechanism is stopped.
Basic Requirements and Limitations

It is to be expected that VecTool, intended for rather unique application requirements, will have some limitations and requirements. The most obvious ones follow:

1. The sensor field of view must be compatible with the tool location. Motion of tool to or along a feature must not block the sensor from seeing the feature (such as an edge or seam).

2. Some of the motion API’s cannot be applied when using VecTool to move tools relative to features. For example, the user would not want to use the blend policy CX_MOVE_THRU for model based moves if the blended path causes the sensor to lose sight of the feature. VecTool must not use a blend policy of CX_MOVE_WAIT because VecTool must be able to stop the mechanism at any time. If CX_MOVE_WAIT is specified, the blend policy will automatically default to CX_MOVE_TO for the VecTool move. The user must follow the move calls with a CxWaitForEnd OfMotion or call one of the signal wait API’s to control the ending of the VecTool motion.

3. VecTools only applies to curvilinear moves (CX_LINEAR_INTERP) and not to joint moves (CX_JOINT_INTERP). VecTool will be disabled for any moves that are joint moves.

4. Generally, the tool and sensor should both be attached to the last independent joint of the robot so that the relative frame pose is constant. Otherwise, it is more difficult for the user to determine tool offsets, based on sensor information.

5. VecTool is meant to be used on 6-axes robots with full pose capability, but can be used on 5-axes robots if the user modifies the sensor offset applied to the tool so that it falls within the constrained orientation capability of the robot (5-axes robots often require the tool motion type CX_Z_POSE or CX_Z_POSE_NO_SPIN).

6. User is responsible for ensuring that tool offsets generated are realistic and not corrupted by measurement noise. The user must also know the scale factors for the sensor being used so that the signal values returned can be used in conjunction with scale factors to generate the offsets which are used in VecTool (for the case where signals are used to store the input from a sensor through an A/D hardware interface).

7. A new API has been provided to change servocard move buffer size to values from 1 to CX_MAX_MOVE_BUFFER (defined as 4). The smaller the buffer size, the more responsive the trajectory will be to sensor offsets. The disadvantage is that the interaction between the CODE server and servocard will be more intense and leave less CPU time for other processes. This means that when VecTool is on, the servocard should only be configured to control one mechanism.

8. The user must be careful in choosing the tool motion type being used, particularly for model based moves. For example, a path move which defines the target (final) frame in some unusual orientation would, for CX_FULL_POSE, interpolate the intermediate moves along the path in such a way as to move the sensor away from the feature being observed. Thus, the user will have to be very careful in defining the target frames for model based moves. The preferred tool motion type
would be **CX_Z_POSE** in which the tool and sensor frames have their Z axes aligned and normal to the feature. This is less of a problem in the dynamic mode since the target frame is being adjusted at each trajectory step.

9. The VecTool capability is currently implemented using the PMAC servocard and requires the CODE trajectory generator. Thus, the trajectory type must be **CX_CALCULATE_TRAJECTORY**.

VecTool should be capable of running at near normal trajectory rates because the custom routines are in-line with the trajectory generator. This means that VecTool should be able to process sensor data at speeds between 80 and 150 Hz for 6-axis robots, depending on the complexity of the robot inverse kinematics, and the user’s routine as organized to access the sensor signals and convert them to tool offsets. The user should remember to select a unique inverse kinematics solution (**CxSetInvkinSoln**) for robots which have multiple inverse kinematics solutions. This will minimize the time spent in the inverse kinematics routines.

**VecTool API’s**

The VecTool API’s can be found in the **Motion Library** of the **Programmer’s Reference Manual**. They are summarized as follows:

```c
long CxSetMoveBufferSize (CxMechanism mech, long buffer_size)
```

*Function* - Sets the VecTool buffer_size. If the value is set to a value greater than **CX_MAX_MOVE_BUFFER**, which as a value of 4, then the buffer size will be set back to its default size.

```c
long CxGetMoveBufferSize (CxMechanism mech, long *buffer_size)
```

*Function* - Gets the VecTool buffer_size

```c
long CxSetVecTool (CxMechanism mech, CxNodeId sensor, long type,long offset_mode)
```

*Function* - Sets the VecTool sensor node, type(CX_VEC_-TOOL_MODEL or CX_VEC_TOOL_DYN), and offset mode (CX_VEC_TOOL_ABS or CX_VEC_TOOL_INCR)

```c
long CxGetVecTool(CxMechanism mech, CxNodeId *sensor,
                 long *type,long *offset_mode)
```

*Function* - Get current VecTool sensor id, type and offset mode

```c
long CxSetVecToolOnOff (CxMechanism mech, long on_off)
```

*Function* - Turns VecTool on or off

```c
long CxGetVecToolOnOff(CxMechanism mech, long *on_off)
```

*Function* - Get current VecTool on or off state
long CxSetVecToolClass (CxMechanism mech, long class)

Function - Sets VecTool class number used to case switch in user_motion.VT

long CxGetVecToolClass (CxMechanism mech, long *class)

Function - Get current VecTool class number

These API’s are described in more detail in the Programmer’s Reference Manual. The section Organizing the Client Process will illustrate the use of some of these API’s.

Implementation

The following provides a simple example of how the user might implement VecTool. The user must first choose a VecTool class number and enter it plus the call to a custom user supplied routine in the custom directory ($ROBTOP/lib/cimetrix/custom) which will display the following files:

cimcont.rc   cimulation.dsp   user_kin.FK
resetric.ac28.dsp cimulation.ico   user_kin.IK
resetric.dsp  robmainNT.c   user_kin.NK
resetric.ico  software_drvr.c   user_motion.AJ
resetric.ipc.dsp  test_drvr.c   user_motion.c
resetric.meidsp.ac28.dsp  threadmain.c   user_motion.FG
resetric.meidsp.dsp  user_cntrl.c   user_motion.JA
resetric.pmac.dsp  user_kin.c   user_motion.JM
resetric.rc   user_kin.CK   user_motion.VT
resetric

test_drvr.c
user_kin.CK
user_motion.VT

Add Custom File to Makefile

The user will create a custom file (any name ok, but we will use my_vec_tool.c here) and custom routine. Inside my_vec_tool.c the user will create a custom VecTool function which we will name my_vec_tool() for illustration purposes. The makefile must be edited by the user so that the routines in my_vec_tool.c can be compiled and linked to make a custom cimserver. Inside the makefile, the user should search for the following code

# Add any new modules to the end of this list.
# Add your custom files here in this section only.
CFILES= PhySignals.c robmain.c threadmain.c user_kin.c \
user_motion.c user_cntrl.c
CXXFILES=

and then modify it to include the custom file (my_vec_tool.c) which will be compiled and linked to make a custom CIMServer:

# Add any new modules to the end of this list.
# Add your custom files here in this section only.
CFILES= PhySignals.c robmain.c threadmain.c user_kin.c \
user_motion.c user_cntrl.c my_vec_tool.c
CXXFILES=

# ==============================================================

Customizing CODE
Next the call to `my_vec_tool()` routine will be added to the include file `user_motion.VT`.

**Add Custom Routine to user_motion.VT**

We choose a VecTool class number $> 0$ (here we select 100) and modify the contents of `user_motion.VT`:

```c
/* Case switch to transfer to user’s VecTool routine */
/* Add your custom case to this file where number should be > 0 */
case 0:
    SetErrorMsg( this_error, CX_TRAJECTORY_ERROR );
    SetErrorStr( this_error, "VecTool class number not set properly;\n    "Set class number > 0 using API CxSetVecToolClass." );
    return(CX_ERROR);
brea;

case 100:
    if(my_vec_tool(q,this_error) == CX_ERROR)
        return(CX_ERROR);
brea;

    /* add your case switch to your VecTool routine something like
    the following; note that the user routine vec_tool() should be
    included in a user file which is added to the makefile
    case 110:
    if(vec_tool(q,this_error) == CX_ERROR)
        return(CX_ERROR);
brea;
    */
```

**Compile and Link VecTool Routine**

The user must simply compile and link the custom routine to make a custom cimserver which can be run from this directory. It is only necessary to type `makefile` in the custom directory for this to happen.

**New Data Structure Members**

New data structure members have been added for VecTool. These can be accessed by the user in the custom VecTool routine and then used by the CODE trajectory generator to provide tool offsets.

*robot data structure additions:*

```c
CxMatrix vec_tool_offset;   /* accumulated vec tool offset */
CxMatrix vec_tool_incr;     /* measured offset deviation */
```

*command_entry data structure additions:*

```c
long vec_tool_on_off;  /* whether current move has VecTool on or off */
long vec_tool_type;    /* CX_VEC_TOOL_MODEL or CX_VEC_TOOL_DYN */
long vec_tool_mode;    /* CX_VEC_TOOL_INCR or CX_VEC_TOOL_ABS */
struct tree_node *sensor_node;     /* pointer to sensor node */
```

*other important data structure members:*

```c
struct tree_node *tcf;      /* pointer to tcf tree_node */
```

The use of these parameters is shown in the programming example which will follow.

**Functional Procedures**

In general, the buffer size should be set to 1 or 2 when applying VecTool. This size should be set in the client process that is used to turn VecTool on. Thus, the buffer size API `CxSetMoveBufferSize()`
Customizing CODE

should be called first. The user must remember to set the buffer back to its default state when not using the VecTool capability. This is done by calling this same API and by entering CX_DEFAULT_BUFFER_SIZE.

If the sensor type uses an A/D hardware interface to enter the sensor data in digital form, then the user will have to modify the signal table so that the sensor used to measure offsets (input through A/D board) correctly maps the sensor input to the desired signal numbers.

The sensor hardware input must be mapped to 16 bit (less precision required) or 32 bit addresses since CODE assigns signals to a long variable (32 bits). Once the user sets up the signal table and wires the sensor properly, the user must perform the necessary calibration to establish the scaling factors which can be used to determine the tool offsets. These calibration constants (along with the appropriate scaling equations) can be included in the custom VecTool function.

The user would normally perform some measurement or calibration to determine the relative pose of the sensor tcf relative to the tool tcf, and enter the relative pose of the tool tcf relative to the sensor tcf into the CODE database (using CIMTools or through API calls). This is easily accomplished by making the tool node a child of the sensor node or vice-versa. The rigid relationship between the sensor and tool node frames makes the application of the tool offset a rather simple process.

Model-Based Procedures

The user would either start some client process or use a teach pendant to position the tool/sensor at some starting configuration. If a client process has not been started, the user would then start the client process(es).

The user would organize the client process to initialize the buffer size to 1 or 2, initialize VecTool using CxSetVecTool and CxSetVecToolClass and then enable VecTool using CxSetVecToolOnOff. Next, the user would call a motion API to move the robot tool relative to some node or path defined in the CODE database (the model node or path). It is assumed that the CODE database rigid-body pose of the part has been adjusted to within some reasonable tolerances of the physical part rigid-body pose.

The user can disable VecTool by stopping the mechanism if it is necessary to stop the motion before the final model target position is reached. Otherwise the move would stop once the model target has been reached. Normally, the user would select a blend policy of CX_MOVE_TO to move past the move command in the client process so that various signal conditions can be used to evaluate the VecTool performance. A CX_MOVE_WAIT in the client process will stop the program at the line of the move until the motion has been completed and not allow the user to stop the mechanism before the move is completed. The user must follow the move calls with a CxWaitForEndOfMotion or call one of the signal wait API’s to control the ending of the VecTool motion.

The API CxSetVecTool is used to set the VecTool type to CX_VEC_TOOL-_MODEL. Model based tracking can be used in either the absolute mode (CX_VEC_TOOL ABS) or incremental mode (CX_VEC_TOOL_INCR) as set by the mode parameter in CxSetVecTool. In the absolute mode the user is responsible for entering the offset matrix for the tool as an absolute offset. If the user does not change the offset, it will continue to be applied as a constant offset.

In contrast, the incremental offset is determined as a deviation from the absolute offset and modifies the absolute offset which is stored.

The only way to stop the move prematurely is to stop the mechanism. The user would not want to set the command_entry data structure member q->vec_tool_on_off to CX_OFF in the custom
VecTool routine, because this would eliminate the application of the absolute offset to the move as the robot slows to a stop. This might move the sensor away from the feature being tracked (depending on the offset currently being applied), plus result in some rather abrupt motion that might lead to a robot joint following error.

Although it is generally recommended that the user not blend moves together using the policy CX_MOVE_THRU, contiguous moves that are tangent to each other could be blended without the sensor losing the feature being tracked.

**WARNING!:** Do not set the parameter q->vec_tool_on_off to CX_OFF in the custom user routine for type CX_VEC_TOOL_MODEL. To stop the mechanism call CxStopMechanism in the client process and then use CxSendMechanismErrorAction to abort or resume the move.

### Dynamic Procedures

The user would either start some client process or use a teach pendant to position the tool/sensor at some starting configuration. If a client process has not been started, the user would then start the client process(es).

The user would organize the client process to initialize the buffer size to 1 or 2, initialize VecTool using CxSetVecTool and CxSetVecToolClass and then enable VecTool using CxSetVecToolOnOff. Next, the user would call the move API CxMoveRelTool to being VecTool in the dynamic mode. VecTool then remains active until it is turned off. Thus, the user can have the mechanism move or remain still as offsets are being generated. CODE will continually update the tool target position and the robot will chase this target until the motion is stopped.

The user would disable VecTool by stopping the mechanism or by setting q->vec_tool_on_off to CX_OFF in the custom user routine when some signal condition is reached that can be tested in the user routine. Normally, the user would use the blend policy CX_MOVE_TO and must avoid using the policy CX_MOVE_WAIT. Since only one CxMoveRelTool move issued for dynamic VecTool, there is no reason to set the blend policy to CX_MOVE_THRU. The user must follow the move calls with a CxWaitForEndOfMotion or call one of the signal wait API's to control the ending of the VecTool motion.

If the user updates the tool target with small offsets, then the robot will not achieve the desired feature following speeds because the distance to accel to full speed is insufficient. The offsets generated by the sensor (or adjusted by the user in the custom routine) must be twice the distance necessary to accelerate to full speed (assuming that acceleration and deceleration settings are the same).

Dynamic tracking can be used in either the absolute mode (CX_VEC_TOOL_ABS) or incremental mode (CX_VEC_TOOL_INCR) as set by the mode parameter in CxSetVecTool. In the absolute mode the user is responsible for entering the offset matrix for the tool as an absolute offset. This offset is used to change the target for the tool dynamically. If the user does not change the offset, it will continue to be applied as a constant offset and cause linear motion of the tool relative to itself. In contrast, the incremental offset is determined as a deviation from the absolute offset and modifies the absolute offset which is stored.

The move can be stopped by calling CxStopMechanism(mech), CxSetVecToolOnOff(mech, CX_OFF) or by turning q->vec_tool_on_off to CX_OFF in the custom VecTool routine. The user can access signal values from both the client process and the
custom user routine to coordinate the dynamic VecTool motion. If the user stops the mechanism by calling CxStopMechanism in the client process, the user can call CxSendMechanism ErrorAction to abort or resume the move.

A VecTool Example
The simple example which follows can be implemented in simulation to demonstrate the use of signals to coordinate a client process with a custom user VecTool routine. Earlier, we identified the custom routine as my_vec_tool() in the file my_vec_tool.c. The custom routine is listed and heavily commented so that the user can follow the logic. This routine has been implemented as VecTool class 100.

Next, the client process is listed and heavily commented. In the client process a move of type CX_VEC_TOOL_MODEL is first made, in the mode CX_VEC_TOOL_INCR. A move of type CX_VEC_TOOL_DYN follows, also in the incremental mode. In between there are some normal moves to a reference target, with VecTool being turned on or off.

The node name for the sensor in the client process is "sensor". The user can access this node in the custom routine using the command_entry data structure parameter q->sensor_node. The user always accesses the tool node in the custom routine as q->tcf.

Organizing the Custom Routine
The routine my_vec_tool() follows. The user should review this routine to understand how VecTool can be implemented. The user should include the following include files to gain access to all necessary parameters.

```
#include <stdio.h>
#include <code/const.h>
#include <code/robconst.h>
#include <code/robererno.h>
#include <code/robot.h>
#include <code/server.h>
#include <code/maketree.h>
#include <code/user_motion.h>
#include <code/matrix.h>
#include <code/sigcalls.h>
#include <code/error.h>

#define CX_X_OFFSET0
#define CX_Y_OFFSET1
#define CX_Z_OFFSET2

/*-------------------vec_tool_sample----------------------------
REMARKS
This is a simple example wherein the sensor is assumed to
only read x, y and z deviations for offsetting the tool. These
deviations are read from the signal table and then converted
to double values. These offsets are then resolved from the
sensor frame to the tcf (tool) frame.
NOTES
* VecTool applies only to curvilinear type moves
*/
```

Customizing CODE
* VecTool requires CX_CALCULATE_TRAJECTORY (CODE trajectory generator)
* If the type is CX_VEC_TOOL_DYN, the user can set
  q->vec_tool_on_off to CX_OFF for any desired user condition
  in the user function and then have the motion complete itself;
  normally you wouldn’t do this for type CX_VEC_TOOL_MODEL.
* If a stop mechanism is issued, the motion will be stopped
  with VecTool still operative if type is CX_VEC_TOOL_MODEL.
  If type is CX_VEC_TOOL_DYN, then VecTool will be turned
  off for the current move and the motion halted.

WRITTEN BY

MODIFICATION BY
Zhaoxue Yang    Aug., 1996

************************ USEFUL SERVER SIDE FUNCTIONS ************************

NODE FUNCTIONS:
void findrm(tree_node *rel_node,tree_node *ref_node,CxMatrix rel_mat)
    Purpose - Given two nodes, determines the relative matrix of rel_node
              relative to ref_node and stores in rel_mat
void findm(tree_node *rel_node,CxMatrix rel_mat)
    Purpose - Given a node, determines the relative matrix of rel_node
              relative to world node and stores in rel_mat

MATRIX FUNCTIONS:
void CxMul4x4 - Same function as available in matrix library of Programmer’s
               Reference Library
void CxMatequ - Same function as available in matrix library of Programmer’s
               Reference Library
void CxInvmat - Same function as available in matrix library of Programmer’s
               Reference Library
void CxRlmat - Same function as available in matrix library of Programmer’s
               Reference Library

NOTE: Refer to matrix library of Programmer’s Reference Library for various
      other matrix library functions which can be used!

SIGNAL FUNCTIONS:
long get_signal(long sig_num,long *value,CxErrorMsg *this_error)
    Purpose - Given signal number sig_num, returns the current value. Returns
              CX_ERROR if server does not know this signal.
long set_signal(long sig_num,long value,CxErrorMsg *this_error)
    Purpose - Given signal number sig_num, sets its value. Returns
              CX_ERROR if server does not know this signal.

----------------------------------------------------------------
long vec_tool_sample(command_entry *q, CxErrorMsg *this_error)
{  
  long x, y, z;  /* signal count */
  double X, Y, Z;  /* offset in physical space */

  /* the calibration ratios from sensor to physical values must be 
  stored in the user routine like the following */

  double CAL_X = 0.2;  /* 0.2 mm per sig count (this example) */
  double CAL_Y = 0.2;  /* 0.2 mm per sig count (this example) */
  double CAL_Z = 0.2;  /* 0.2 mm per sig count (this example) */

  CxMatrix rel_mat;  /* rel matrix of sensor rel to tcf */
  CxVector offset;  /* vector of xyz offsets read by sensor */
  robot *rob;

  rob = q->rob;

  /* the sensor signals must be identified in the signal table and 
  correctly wired to the sensor through the A/D board; in this 
  example the signals were set in a client process as software 
  signals to test the VecTool process and motion algorithms */

  if(get_signal(CX_X_OFFSET, &x, this_error) == CX_ERROR) 
    return(CX_ERROR);

  if(get_signal(CX_Y_OFFSET, &y, this_error) == CX_ERROR) 
    return(CX_ERROR);

  if(get_signal(CX_Z_OFFSET, &z, this_error) == CX_ERROR) 
    return(CX_ERROR);

  /* perform the conversion to physical space */

  X = CAL_X * ((double) x);
  Y = CAL_Y * ((double) y);
  Z = CAL_Z * ((double) z);

  /* now load offset vector with X, Y, Z offsets */

  offset[0] = X;
  offset[1] = Y;
  offset[2] = Z;

  /* get frame of sensor rel to tcf and store in rel_mat */

  findrm(q->sensor_node, q->tcf, rel_mat);

  /* apply offsets to tcf frame */

  CxVecmlt(CX_TRUE, rel_mat, offset, offset);

  /* now load the offsets; if mode is INCR then load vec_tool_incr array 
  and the vec_tool_offset array will be automatically updated with 
  the offset values; if mode is absolute then the user must change 
  the vec_tool_offset array if it is to be modified */

  if(q->vec_tool_mode == CX_VEC_TOOL_INCR)  /* set only vec_tool_incr */
  {
    rob->vec_tool_incr[3][0] = offset[0];
    rob->vec_tool_incr[3][1] = offset[1];
    rob->vec_tool_incr[3][2] = offset[2];
  }
  else  /* CX_VEC_TOOL_ABS: set vec_tool_offset */
  {
    rob->vec_tool_offset[3][0] = offset[0];
    rob->vec_tool_offset[3][1] = offset[1];
    rob->vec_tool_offset[3][2] = offset[2];
  }  
}
Note that the routine is organized so that both the MODEL and DYN types can be commanded as well as the ABS and INCR modes for either type. If the INCR mode is used, the user must simply modify the incremental offset matrix rob->vec_tool_incr by the appropriate offsets as resolved in the tcf (tool) frame. CODE will then apply the incremental offsets to the current rob->vec_tool_offset to get a new rob->vec_tool_offset. In the ABS mode the user is responsible for applying the offsets to generate directly a new rob->vec_tool_offset, otherwise, the last value will be continually applied as the mechanism moves.

Organizing the Client Process
A simple client process developed for this user VecTool function follows:

```c
#include <stdio.h>
#include <code/const.h>
#include <code/robconst.h>
#include <code/clients.h>
#include <code/robpac.h>

/* define signals used in the custom VecTool routine */
#define         CX_X_OFFSET     0
#define         CX_Y_OFFSET     1

main( void )
{
    CxServer Server;
    CxMechanism mech;

    CxNodeId rob, tcf, t1, t2, sensor;
    long type, mode, class, on;

    /************* set up the workcell ************/
    Server = CxOpenServer("ered", CX_SYSTEM_V, 0);
    CxGetNamedNodeIds( Server, "rt3200", &rob );
    CxGetNamedNodeIds( Server, "t1", &t1);
    CxGetNamedNodeIds( Server, "t2", &t2);
    CxGetNamedNodeIds( Server, "sensor", &sensor);
    CxGetNamedNodeIds( Server, "tcf", &tcf);

    mech = CxOpenMechanism( Server, rob, CX_CONTROL );

    CxSetSimrate( Server, 0.01);
    CxSetTrajectoryRate( mech, 0.01);
    CxSetAccelType( mech, CX_CONST_RAMP_TIME);
    CxSetTrapAccelTimes( mech, 0.5, 0.5);
    CxSetTrajectoryMode( mech, CX_CALCULATE_TRAJECTORY );
    CxSetTipSpeed(mech, 100.);
```

10/98 6-11
/* VecTool only applies to curvilinear, not joint moves */

CxSetInterpType(mech,CX_LINEAR_INTERP);
CxSetBlendPolicy(mech,CX_MOVE_WAIT);

/* move to a reference target */

CxMoveToNode(mech,t1,tcf);

/* the user must be very careful with the tool motion type, else the tool orientation will cause the sensor to be moved to a non-viewing pose; this is a particular problem when the type is CX_VEC_TOOL_MODEL when the tool is commanded to follow a path between two frames. */

CxSetToolMotionType(mech,CX_FIXED_ORIENT);

/* the user must identify the class number so that user_vec_tool() can case switch to the call of the custom VecTool routine; presently CIMTools does not provide utilities to specify this class number */

CxSetMoveBufferSize( mech, 2);
CxSetVecToolClass( mech, 100);

/* set VecTool to type DYN and mode INCR */

CxSetVecTool(mech,sensor,CX_VEC_TOOL_DYN,CX_VEC_TOOL_INCR);
CxSetVecToolOnOff(mech,CX_ON);

/* these calls are here to check the get routines */

CxGetVecToolClass(mech,&class);
CxGetVecTool(mech,&sens,&type,&mode);
CxGetVecToolOnOff(mech,&on);

/* in this case we set some offsets so that the target will be adjusted dynamically; in the physical application the sensor will be read inside the custom VecTool routine */

CxSetSignal(Server,CX_X_OFFSET,400);
CxSetSignal(Server,CX_Y_OFFSET,0);

/* for type DYN the user must start the move artificially with some small move (say in the range of 1 - 10; the sensor will then be read in the trajectory loop and the target adjusted continually as the tcf (tool) chases the target (sensor) */

CxSetBlendPolicy(mech,CX_MOVE_TO);
CxMoveRelTool(mech,tcf,"XYZ",0.,0.,0.,0.,0.,0.);
CxDelay(1,0); /* force a delay of one second */

/* force the dyn move to end */

CxSetSignal(Server,CX_X_OFFSET,0);

/* turn VecTool off and return to t1 target */

CxSetVecToolOnOff(mech,CX_OFF);
CxSetMoveBufferSize(mech,CX_DEFAULT_BUFFER_SIZE);
CxSetBlendPolicy(mech, CX_MOVE_WAIT);
CxMoveToNode(mech, t1, tcf);

/* now do a move of type MODEL in INCR mode */

CxSetMoveBufferSize(mech, 2);
CxSetVecTool(mech, sensor, CX_VEC_TOOL_ABS, CX_VEC_TOOL_INCR);
CxSetVecToolOnOff(mech, CX_ON);
CxSetBlendPolicy(mech, CX_MOVE_TO);
CxSetSignal(Server, CX_Y_OFFSET, -1);
CxMoveToNode(mech, t2, tcf); /* move will stop when get to t2 */

CxSetMoveBufferSize(mech, CX_DEFAULT_BUFFER_BUFFER_SIZE);

CxRobpacExit();
}

The user will normally set the move buffer size to 2 to minimize the latency between the sensor adjustment and the actual tcf response. The maximum buffer size is CX_MAX_MOVE_BUFFER_SIZE (which is set at 4).

**Warnings**

VecTool procedures provide flexibility for the user to change dynamically the tool pose in response to sensor offsets as the mechanism moves the tool along some feature. These offsets are applied in addition to offsets which are commanded in the normal move API’s. For example, if the user uses CxMoveRelNode to move relative to some target with non-zero offsets, and, in addition, a sensor determines offsets to be applied, these offsets will be applied after the CxMoveRelNode offsets are first applied. The potential for user error in applying these offsets is reduced if the user minimizes the need for normal motion offsets. For example, it would be better to redefine tool or target nodes so that a direct move to the target is commanded rather then a move to a target with some offset.

VecTool obviously has the potential of causing robot collisions because of user error in programming or bad sensor data. To minimize such errors, the robot will slow itself down if it approaches joint limits under certain speed and acceleration conditions that would cause the mechanism to exceed its joint limits.

Simulation should be used to prove the user custom interfaces and to test the client processes before actually bringing VecTool online. The user can modify signals in a client process to simulate the hardware signals that would be read from the sensor. These signals could then be used to move the mechanism in simulation and provide a test of the programming integrity.

User errors in custom routine programming can cause fatal errors in the CODE server, and cause system crashes. Thus, the user must take all precautions to not overwrite data structure arrays, and other common fatal programming errors.
Index

A
adding
  driver modules to makefile .................................. 4-40
  libraries to makefile .......................................... 4-40
  aligning tools to the z axis ................................ 3-33
arguments, robot data structure ................................ 3-14
auto DH .................................................................... 3-3

B
base joint (primal).................................................... 3-1
blend_policy ................................................................ 5-11
  see motion data structures, parameters, comma... 5-11
  values
    MOVE_WAIT .......................................................... 5-13
    values ................................................................ 5-13
    MOVE_THRU .......................................................... 5-13
    MOVE_TO ............................................................ 5-13
    blend_status ......................................................... 5-11
    see command_entry .............................................. 5-11
    blend_status values
      DO_NOT_BLEND .................................................. 5-13
      IN_BLEND .......................................................... 5-13
      NOT_IN_BLEND .................................................. 5-13
  branch_custom_invkin, defined............................... 3-2
  branch_DH_invkin, defined...................................... 3-2
  branching robots, using the IMCCIMServer with ....... 3-32

cartesian path motion
  CODE default trajectory routines........................... 5-6
  rate checking ....................................................... 5-6
  cell model, accessing ........................................... 4-1
CIMServer
  Non-Threaded version ............................................ 4-5
  Threaded version .................................................. 4-5
  CODE acceleration control modes........................... 5-4
  CONST_RAMP_TIME ................................................ 5-4
  CODE acceleration control modes
    CONST_RAMP_ACCEL ............................................. 5-4

C

D
data structures
  command_entry ..................................................... 4-12
  mechanism_entry .................................................. 4-10
  of importance ..................................................... 4-5
  signals_t and signal_drivers_t ............................... 4-6, 4-8
  debugging invkin routines ..................................... 3-11, 3-21
  defining custom exact inverse kinematics ............... 3-22
  defining signal drivers ....................................... 1-4
  defining signals .................................................. 1-6
  defining the device name ..................................... 1-5
  degrees-of-freedom (dof) ....................................... 3-1
  see also kinematics ............................................. 3-1
  Denavit-Hartenberg .............................................. 3-4
  dependent joints, defined ..................................... 3-1
  developing custom device interfaces, steps to......... 4-1
  device and I/O interface functions ......................... 4-21
  CODE curve, defined............................................. 5-3
  CODE implementation of trajectory generator ............ 5-24
  CODE include file user_motion modification ............... 5-23
  CODE kinematic interfaces ..................................... 3-2
  CODE trajectory control ......................................... 6-3
  command_entry ..................................................... 4-12
  complex signals, defining...... 1-9, 1-13, 1-15, 1-17, 1-18
  constructing branching robots ................................ 3-33
  controlling workcell activities ............................. 4-1
  creating a signal table ....................................... 1-4
  curvilinear speed ............................................... 5-7
  path lengths ........................................................ 5-7
  custom device controller interface, integrating ....... 4-2
  custom exact inverse kinematics ............................. 3-22
  defining ........................................................... 3-22
  custom inverse kinematics, defined ......................... 3-4
  custom motion interfaces ........................................ 5-1, 6-1
  custom trajectory generator implementation .............. 5-23
  modification of CODE include file ......................... 5-23
  custom trajectory generator .................................... 5-21
  addition of function ............................................ 5-25
  compilation ....................................................... 5-28
  generation .......................................................... 5-26
  limitations .......................................................... 5-25
  theory behind ...................................................... 5-22
  custom user joint rate routine
    CODE implementation ........................................... 5-38
    compila.......................................................... 5-39
  custom user joint rate routine ................................ 5-37
  CODE implementation
    addition .......................................................... 5-38
    modifica.......................................................... 5-38
    theory behind .................................................... 5-37

6-14  Customizing CODE
device interfacing................................. 4-1
architecture........................................ 4-1
developing device interfaces.................. 4-4
methodology of ..................................... 4-1
device name, defining......................... 1-5
DH invKin routines.............................. 3-2
DH parameter file format..................... 3-5
DH parameters
 calculating....................................... 3-4
 example file................................. 3-7, 6-7
 required parameters (minimum)............ 3-5
DH solutions
 identifying unique ......................... 3-8
 integrating ..................................... 3-14
driver implementation methods .......... 4-2
driver instance, setting...................... 1-5
driver modules
 adding to makefile.......................... 4-40
driver type, setting ......................... 1-4
drivers
 defining in signal table .................. 1-4

E
entering numerical routines with CODE...... 3-26
equation for unknown joint values .......... 3-8
exact inverse kinematics .................... 3-2
equation for unknown ............................................. 3-8
example of a custom trajectory generator
 see custom trajec ............................ 5-21
example of a custom trajectory generator 5-21
equation of a custom user joint rate routine
 see custom use........................................ 5-37
equation of a custom user joint rate routine 5-37
equations for integrating DH solutions ... 3-14

F
class number
 defined ........................................... 3-3
specifying for robot ............................ 3-31
forward kinematics
 classification number, defined
 see also ............................................. 3-3
 introduction ...................................... 3-1
routine, func_dep_jnt .......................... 3-29
defined............................................. 3-3
functionally dependent joints............... 3-29
functions
device and I/O ................................... 4-21
motion ............................................. 4-21

G
generating desired velocity profile......... 5-22
generating DH parameter set
 see also auto DH.............................. 3-3
generating inverse kinematics............. 3-10

H
hints for integrating DH solutions ......... 3-14

I
iclass number..................................... 3-2, 3-8
identifying unique DH solution .......... 3-8
initialization routines ....................... 5-19
inner tcfs ........................................ 5-10
integrating custom device controller interfaces 4-2
integrating DH solutions, examples and hints 3-14
integrating inverse kinematics .......... 3-10
interface functions
 device and I/O .................................. 4-21
motion ............................................. 4-21
interface routines........................... 3-2, 3-3
inverse kinematics
 auto DH......................................... 3-3
custom exact ..................................... 3-22
debugging solutions ........................ 3-21
exact............................................... 3-2
exact invkin solutions ........................ 3-2
generating ........................................ 3-10
integrating ....................................... 3-10
introduction
 see also invkin .................................. 3-1
mathematical procedures for ............. 3-7
numerical ......................................... 3-2, 3-25
specifying types ................................ 3-2
zero state
 defined .......................................... 3-3
invkin
 classification number
 see also iclass number .................... 3-2
custom exact ..................................... 3-22
debugging routines ......................... 3-11, 3-21
exact invkin solutions ................. 3-2
introduction ..................................... 3-1
numerical ......................................... 3-25
see also inverse kinematics .............. 3-1
specifying types ................................ 3-2
invkin solutions
 library in CODE ............................... 3-3

J
ejerk, defined .................................... 5-22
joint data structure .......................... 5-16
joint motion (JOIN_TINTERP) .......... 5-4
joint types
 mixing ............................................ 3-27
joint values
 equation for unknown ....................... 3-8
scaling ........................................... 3-27

K
kinematics ........................................ 3-1
DH inverse kinematics ..................... 3-4

Customizing CODE
exact inverse kinematics .............................................. 3-3
functionally dependent joints ........................................... 3-29
interface routines ...................................................... 3-2, 3-3
kinematic interfaces ..................................................... 3-2

libraries
adding to makefile .................................................... 4-40
logical device name
see also device name ................................................... 1-5
logical name, signal ..................................................... 1-6

M
maintaining curvilinear speed ......................................... 5-7
mechanism_entry ......................................................... 4-10
mixing joint types ....................................................... 3-27
modification, CODE include file user_motion.fg ........... 5-23
motion blending .......................................................... 5-3
MOVE_THRU .................................................. 5-3
MOVE_TO .................................................. 5-3
MOVE_WAIT .................................................. 5-3
motion concepts .......................................................... 5-7
motion data structures
parameters .............................................................. 5-10
motion data structures ................................................ 5-10
parameters
command_entry .......................................................... 5-11
blend_pol .............................................................. 5-11
blend_sta .............................................................. 5-11
motion_st .............................................................. 5-11
ramp_stat .............................................................. 5-11
type ................................................................. 5-11
mechanism_entry ....................................................... 5-11
motion interface functions .......................................... 4-21
motion interfaces
routines
user_trajectory .......................................................... 5-17
motion interfaces ...................................................... 5-17
routines .............................................................. 5-17
user_convert_act_to_jnt .............................................. 5-17
user_convert_jnt_to_act .............................................. 5-17
user_joint_rates ..................................................... 5-17
motion_status .......................................................... 5-11
see motion data structures, parameters, comm .... 5-11
values
IN_MOTION .................................................... 5-12
motion_status
values ................................................................. 5-12
END_OF_MOTION .................................................. 5-12
NO_MOTION ...................................................... 5-12
values
IN_MOTION ...................................................... 5-12

N
naming the signal table .................................................. 1-4

NC_ROBOT
see branching robots .................................................. 3-32
non-threaded Server, defined ........................................... 4-5
numerical approach ..................................................... 3-26
umerical inverse kinematics ........................................... 3-26
entering routines ....................................................... 3-26
numerical invkin
problems and solutions .............................................. 3-28
numerical solution approach to kinematics .................. 3-26

P
planning workcell activities .......................................... 4-1
pose, defined .......................................................... 3-1
problems with numerical invkin .................................... 3-28

R
ramp_status ............................................................ 5-11
see motion data structures, parameters, comm .... 5-11
values
FALL .............................................................. 5-12
RISE .............................................................. 5-12
STEADY ............................................................ 5-12
ramp_status
values ................................................................. 5-12
reclass number .......................................................... 3-26
defined .............................................................. 3-3
reconfiguring a robot, methods ................................... 3-1
redundant, defined ................................................... 3-3
redundant/numerical invkin classification number,
defined
see ................................................................. 3-3
required DH file format ............................................. 3-16
ROBLINE Server ..................................................... 4-1
robot
reconfiguring .......................................................... 3-1
robot base frame ..................................................... 3-6, 3-7
robot data structure .................................................. 5-15
robot data structure, arguments ................................... 3-14

S
saving DH parameters .................................................. 3-8
letter sequence .......................................................... 3-8
scaling joint values ..................................................... 3-27
see also kinematics ..................................................... 3-1
setting the driver instance ......................................... 1-5
setting the driver type ................................................. 1-4
signal header file, creating .......................................... 1-26
signal table
creating .............................................................. 1-4
defining drivers ....................................................... 1-4
defining signals ....................................................... 1-6
defining the device name ........................................... 1-5
driver type, setting ..................................................... 1-4
naming ................................................................. 1-4
setting logical name, signal ........................................ 1-6
setting the driver instance ........................................... 1-5
signal types ............................................................ 1-7
signal tables ........................................... 1-1, 2-1, 2-2, 2-3
signal types, specifying ................................. 1-7
signal_drivers_t ............................................. 4-6, 4-8
signals
complex .............................................. 1-9, 1-13, 1-15, 1-17, 1-18
  simple .................................................. 1-9
signals_t .................................................. 4-6, 4-8
simple signals, defining ................................... 1-9
specifying fclass number for robot ................. 3-31
speed, maintaining along path segments
  see also curvilinea ......................................... 5-7
starting CODE, command line ...................... 1-1
steps to developing custom device interfaces ....... 4-1
symmetric interface
  planning and controlling workcell with ........... 4-1

T
tcf
defined .................................................. 3-2
TCF
  see tool control frame ................................. 5-9
TCP
  see also tool control frame ....................... 5-9
TCP .......................................................... 5-9
terminal control frame (tcf)
  see also tcf .............................................. 3-2
terminal joint (distal) .................................... 3-1
threaded Server, defined ................................. 4-5
tool control frame ........................................ 5-9
tool motion types
  Z_POSE_NO_SPIN ........................................... 5-8
tool motion types .......................................... 5-7
  FIXED_ORIENT ............................................. 5-9
  FULL_POSE ................................................... 5-8
  Z_POSE ....................................................... 5-8
tool offsets .............................................. 5-9
tools, aligning to z axis .................................. 3-33
trajectory generator ...................................... 6-4
transformation operations .............................. 3-7
type ......................................................... 5-11
  see motion data structures, parameters,
    command_entry ..................................... 5-11
type= ROB MOVE ........................................... 5-13

U
unknown joint values, equation for ................... 3-8
user_convert_act_to_jnt .................................. 5-21
user_convert_jnt_to_act .................................. 5-21
user_joint_rates ......................................... 5-21
user_trajectory .......................................... 5-18

V
velocity, generate desired profile ........................ 5-22

Z
z axis, aligning tools .................................... 3-33
zero state, defined ....................................... 3-3