- NOTICE -
This document was written to accompany CODE version 4.1. 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 - 1999 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.

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, CIMCal, CIMApp and CIMTeach 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.

Part Number (CODE-API-4.1.0 NT V1 05/99)
C Function Prototypes

Controller Management

- long CxCloseMechanism(CxMechanism mech_to_close)
- long CxCloseServer(CxServer Server_to_close)
- long CxDetach(void)
- long CxDisableAmps(CxMechanism mech)
- long CxEnableAmps(CxMechanism mech)
- long CxGetControlMode(CxMechanism mech, long *mode)
- long CxGetControllerAttr(CxMechanism mech, long *attr_mask)
- CxController CxGetControllerFromMech(CxMechanism mech)
- CxController CxGetControllerFromSignal(CxServer Server, long signal)
- long CxGetControllerStatus(CxMechanism mech, long stat_mask, long *ret_stat_mask)
- long CxGetControllerType(CxServer Server, CxNodeId mech_id, long *type)
- long CxGetCycleTime(CxServer Server, double *time)
- char CxGetRoblineServerHost(void)
- char CxGetSecurityHost(void)
- long CxGetServerType(CxMechanism mech, long *type)
- long CxGetServerVersion(CxServer server, char* server_version)
- long CxGetSimrate(CxServer Server, double *rate)
- long CxGetTrajectoryMode(CxMechanism mech, long *mode)
- long CxGetTrajectoryOverride(CxMechanism mech, long *trajectory_override)
- long CxHomeMechanism(CxMechanism mech)
- bool CxIsServerReady(char* ServerName, long protocol)
- CxMechanism CxOpenMechanism(CxServer Server, CxNodeId mech_id, unsigned long mode)
- CxServer CxOpenServer(char *server_name, long protocol, long tool_or_app)
- void CxRobpacExit(void)
- long CxSendDeviceCommand(CxController cntrl, long queued, char *cmd_str, long size_of_cmd, char *ret_str, long *size_of_ret)
- long CxSetControllerType(CxServer Server, CxNodeId mech_id, long type)
- long CxSetCycleTime(CxServer Server, double time)
- long CxSetRoblineServerHost(char *hostname)
- long CxSetSecurityHost(char *hostname)
long CxSetServerType (CxMechanism mech, long type)
long CxSetSimTrajectoryMode (CxServer Server, long mode)
long CxSetSimrate (CxServer Server, double rate)
long CxSetTrajectoryMode (CxMechanism mech, long trajectory_mode)
long CxSetTrajectoryOverride (CxMechanism mech, long trajectory_override)
long CxStopMechanism (CxMechanism mech)
bool CxWaitForServerReady (char* ServerName, long protocol, DWORD timeout)

Motion Control

long CxClearStartSignal (CxMechanism mech)
long CxClearStopSignal (CxMechanism mech)
long CxConstVelMoveAllAxes (CxMechanism mech, double *axes_vel)
long CxConstVelMoveSingleAxis (CxMechanism mech, long axis, double axis_vel)
long CxDeleteConfig (CxMechanism mech, const char *name)
long CxGetAccelType (CxMechanism mech, long *type)
long CxGetActualInterpType (CxMechanism mech, CxNodeId node, long *type)
long CxGetActualInvkinSoln (CxMechanism mech, CxNodeId node, long *soln)
long CxGetActualJointSpeed (CxMechanism mech, CxNodeId node, double *speed)
long CxGetAxes (CxMechanism mech, double *axes_values)
long CxGetBlendPolicy (CxMechanism mech, long *policy)
long CxGetBlendType (CxMechanism mech, long *type)
long CxGetConfigName (CxMechanism mech, long index, char *name)
long CxGetInnerTcfOffset (CxMechanism mech, CxNodeId node, char axes[4], CxVector angles, CxVector vec)
long CxGetInterpType (CxMechanism mech, long *interp_type)
long CxGetInvkinSoln (CxMechanism mech, long *soln_num)
long CxGetJacobian (CxMechanism mech, long *flag)
long CxGetJointMoveTime (CxMechanism mech, CxTimedMove *move, double *dofinitial[CX_MAXJOINTS], double *dofinal[CX_MAXJOINTS], long *jnt_slow, double *move_time)
long CxGetJointSpeed (CxMechanism mech, double *speed)
long CxGetLatchedAxes (CxMechanism mech, double *value)
long CxGetMotionStatus (CxMechanism mech, long *mstat)
long CxGetMotionTrack (CxMechanism mech, long *flag)
long CxGetMoveBufferSize (CxMechanism mech, long *size)
long CxGetMoveTangent (CxMechanism mech, double *tang_value)
long CxGetNumConfig (CxMechanism mech, long *num)
long CxGetPercentBlendSpeed (CxMechanism mech, double *fraction)
long CxGetSAccelTimes (CxMechanism mech, double *rise_S_time, double *fall_S_time)
long CxGetScrewSpeed (CxMechanism mech, double *speed)
long CxGetTipSpeed (CxMechanism mech, double *speed)
long CxGetToolMotionType (CxMechanism mech, long *tool_motion_type)
long CxGetToolName (CxMechanism mech, CxNodeId node, char *name)
long CxGetToolSpeedVec (CxMechanism mech, CxNodeId node, CxVector angles, CxVector vec)
long CxGetTrajAccelType (CxMechanism mech, long *type)
long CxGetTrajectoryRate (CxMechanism mech, double *rate)
long CxGetTrapAccelRamps (CxMechanism mech, double *rise_ramp, double *fall_ramp)
long CxGetTrapAccelTimes (CxMechanism mech, double *rise_time, double *fall_time)
long CxGetTrapScrewAccel (CxMechanism mech, double *accel, double *decel)
long CxGetWeaveOnOff (CxMechanism mech, long *on_or_off)
long CxGetWeaveType (CxMechanism mech, long *type, double *ampl, double *freq)
long CxICanMove (CxMechanism mech, CxNodeId target, CxNodeId tcf, char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxLatchOnTrigger (CxServer Server, long latch_signal, CxMechanism mech, long trigger_flag)
long CxListConfig (CxMechanism mech)
long CxLoadMoveParameters (CxMechanism mech, CxNodeId rob_id, CxTimedMove *move)
long CxMoveAllAxes (CxMechanism mech, double *axes_values)
long CxMoveAway (CxMechanism mech, CxNodeId tool, double z)
long CxMoveNearNode (CxMechanism mech, CxNodeId target, CxNodeId tool, double z)
long CxMoveRelArc (CxMechanism mech, CxNodeId target, CxNodeId tcf, long path_axis, char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxMoveRelArc3 (CxMechanism mech, CxNodeId target, CxNodeId tcf, CxNodeId mid_arc, char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxMoveRelArchNormal (mechanism mech, node_id ref_node, node_id tcf, double pt_on_center, double arc_normal, double arc_angle)
long CxMoveRelNode (CxMechanism mech, CxNodeId target, CxNodeId tcf, char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxMoveRelPath (CxMechanism mech, CxNodeId target, CxNodeId tool, char start_seg[CX_MAXNAME], end_seg[CX_MAXNAME], char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxMoveRelPathFrame (CxMechanism mech, CxNodeId path, CxNodeId tool, char path_seg[CX_MAXNAME], char axes[4], double a1, double a2, double a3, double x, double y, double z)
long CxMoveRelTool(CxMechanism mech, CxNodeId tool, char axes[4],
        double a1, double a2, double a3, double x, double y, double z)
long CxMoveSingleAxis(CxMechanism mech, long axis, double axis_value)
long CxMoveToConfig(CxMechanism mech, const char *config_name)
long CxMoveToNode(CxMechanism mech, CxNodeId target, CxNodeId tcf)
long CxReteachNode(CxMechanism mech, CxNodeId node, CxNodeId tcf)
long CxSafeJumpMove  (CxMechanism mech, long Z_axis_joint_number, CxNodeId
        target_base, CxNodeId tcf, double x, double y, double z, double
        theta, CxNodeId safe, double trigger_offset, double complete_offset
 )
long CxSetAccelType(CxMechanism mech, long type)
long CxSetBlendPolicy(CxMechanism mech, long policy)
long CxSetBlendType(CxMechanism mech, long type)
long CxSetInterpType(CxMechanism mech, long interp_type)
long CxSetInvkinSoln(CxMechanism mech, long soln_num)
long CxSetJacobian(CxMechanism mech, long flag)
long CxSetJointMoveTime(CxMechanism mech, long stop_on_error, double
        move_time)
long CxSetJointSpeed(CxMechanism mech, double speed)
long CxSetMotionTrack(CxMechanism mech, long flag)
long CxSetMoveBufferSize(CxMechanism mech, long size)
long CxSetMoveTangent(CxMechanism mech, double tang_value)
long CxSetPathSegAccel  (CxServer server, CxNodeId node, char *seg_name,
        double accel);
long CxSetPathSegmentSignal  (CxServer server, CxNodeId node, char
        *seg_name, long start_signal, long start_sig_value, long end_signal,
        long end_sig_value);
long CxSetPathSegVelocity  (CxServer server, CxNodeId node, char
        *seg_name, double velocity);
long CxSetPercentBlendSpeed(CxMechanism mech, double fraction)
long CxSetSAccelTimes(CxMechanism mech, double rise_S_time, double
        fall_S_time)
long CxSetScrewSpeed(CxMechanism mech, double speed)
long CxSetStartSignal(CxMechanism mech, long start_signal, long
        sig_value)
long CxSetStopSignal(CxMechanism mech, long stop_signal, long
        sig_value)
long CxSetTipSpeed(CxMechanism mech, double speed)
long CxSetToolMotionType(CxMechanism mech, long tool_motion_type)
long CxSetTrajAccelType(CxMechanism mech, long type)
long CxSetTrajectoryRate(CxMechanism mech, double rate)
long CxSetTrapAccelRamps(CxMechanism mech, double rise_ramp, double
        fall_ramp)
long CxSetTrapAccelTimes(CxMechanism mech, double rise_time, double
        fall_time)
long CxSetWeaveOnOff(CxMechanism mech, long on_or_off)
long CxSetWeaveType(CxMechanism mech, long type, double weave_ampl, double weave_freq);
long CxTeachConfig(CxMechanism mech, const char *name, long over_write, long serve_type);
long CxTeachNode(CxMechanism mech, const char *name, CxNodeId parent, CxNodeId tcf);
long CxTeleMove(CxMechanism mech);
long CxUpdateJoints(CxMechanism mech);
long CxUpdateTcf(CxMechanism mech, CxNodeId tcf, CxNodeId update_ele, char axes[4], CxVector angles, CxVector vec);
long CxWaitForEndOfCommandedMotion(CxMechanism mech);
long CxWaitForEndOfMotion(CxMechanism mech);

Events, States, and I/O

long CxAsyncPulseSignal(CxServer Server, long signal_num, long value, long time);
void CxDelay(long sec, long microsec);
long CxDisablePosComp(CxMechanism mech, long axis);
long CxDisableWhen(CxServer Server, long when_signal, long when_value, long CxSetSignal, long set_value);
long CxGetSignalId(CxServer Server, char *name, long *index);
long CxGetSignalName(CxServer Server, long index, char *name);
long CxGetSignalValue(CxServer Server, long signal_num, long *value);
long CxInterruptOnChange(CxServer Server, long signal_num, CxMechanism mech, void (*handler_proc)());
long CxInterruptOnChangeOff(CxServer Server, long signal_num);
long CxInterruptOnLowerThreshold(CxServer Server, long signal_num, long value, CxMechanism mech, void (*handler_proc)());
long CxInterruptOnLowerThresholdOff(CxServer Server, long signal_num);
long CxInterruptOnUpperThreshold(CxServer Server, long signal_num, long value, CxMechanism mech, void (*handler_proc)());
long CxInterruptOnUpperThresholdOff(CxServer Server, long signal_num);
long CxInterruptOnValue(CxServer Server, long signal_num, long value, CxMechanism mech, void (*handler_proc)());
long CxInterruptOnValueOff(CxServer Server, long signal_num, long value);
long CxPositionComp(CxMechanism mech, long axis, long num_pos, double *comp_pos, long status_sig, long output_sig, long output_value, double width, long mode, double rollover);
long CxPulseSignal(CxServer Server, long signal_num, long value, long time);
long CxQueuedSetSignal(CxMechanism mech, long signal_num, long value);
long CxQueuedWaitForSignal(CxServer Server, long signal_num, long value, long type, long timeout);
long CxSetScanRate(CxServer Server, long scan_rate);
long CxSetSignal(CxServer Server, long signal_num, long value);
long CxWaitForChange(CxServer Server, long signal_num, long *value, long timeout)
long CxWaitForLowerThreshold(CxServer Server, long signal_num, long *value, long timeout)
long CxWaitForUpperThreshold(CxServer Server, long signal_num, long value, long timeout)
long CxWaitForValue(CxServer Server, long signal_num, long value, long timeout)
long CxWhen(CxServer Server, long when_signal, long when_value, long CxSetSignal, long set_value)

Exception Handling

Introduction - an overview of exception (error) handling

void CxAssignErrorHandler(CxServer Server, long (*Error Handler)(CxServer Server, CxMechanism Mechanism, int tag, int async, char *err_buff))

CxErrorStruct *CxGetAsyncError (CxServer server)
long CxGetAsyncErrorAction (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorConstant (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorFile (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorId (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorJoint (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorLine (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorMech (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorMotor (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorNumber (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorTime (CxGetErrorStruct *pErrorStruct)
long CxGetAsyncErrorTraceCount (CxGetErrorStruct *pErrorStruct)

CxErrorStruct *CxGetError (CxServer server)
long CxGetErrorAction (CxGetErrorStruct *pErrorStruct)
long CxGetErrorConstant (CxGetErrorStruct *pErrorStruct)
long CxGetErrorFile (CxGetErrorStruct *pErrorStruct)
long CxGetErrorId (CxGetErrorStruct *pErrorStruct)
long CxGetErrorJoint (CxGetErrorStruct *pErrorStruct)
long CxGetErrorLine (CxGetErrorStruct *pErrorStruct)
long CxGetErrorLogPolicy (CxServer server, long *mask, char *f_name, long *append)
long CxGetErrorMech (CxGetErrorStruct *pErrorStruct)
long CxGetErrorMotor (CxGetErrorStruct *pErrorStruct)
long CxGetErrorMsg (CxGetErrorStruct *pErrorStruct)
long CxGetErrorNumber (CxGetErrorStruct *pErrorStruct)
void CxGetErrorPolicy (CxServer server, long *policy)
long CxGetErrorTime (CxGetErrorStruct *pErrorStruct)
long CxGetErrorTraceCount (CxGetErrorStruct *pErrorStruct)
long CxSendMechanismErrorAction (CxMechanism mech, long action)
void CxSetErrorLogPolicy (CxServer server, long mask, char *f_name, long append
long CxSetErrorPolicy (CxServer server, long policy)
Controller Management
**CxCloseMechanism**

Closes a mechanism previously opened for either control or monitor

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxCloseMechanism (CxMechanism mech_to_close)
```

**ARGUMENTS**

- `mech_to_close` Mechanism ID of the mechanism to be closed.
  - CxMechanism ID returned by a call to CxOpenMechanism

**DESCRIPTION**

This function closes a mechanism and discontinues control or monitoring. If the process opened the mechanism for `CX_CONTROL` and the mechanism is in motion, the motion will stop. Any error conditions will also be cleared, and the motion queue on the CIMServer corresponding to the mechanism is flushed (see CxSendMechanismErrorAction). In order to finish any motions pending in the CIMServer's motion queue, the CxWaitForEndOfMotion API function should be called before calling this function. CxCloseMechanism does not have to be called if CxRobpacExit is invoked. CxRobpacExit calls CxCloseMechanism and CxCloseServer internally.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Specified mechanism ID does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a server and a mechanism, then sets the error policy and error log policy to write errors to a file. Next, it homes the mechanism. (Note: In simulation mode CxHomeMechanism has no effect). Then it moves different joints of the mechanism. Once all the moves are complete, it closes the mechanism and closes its connection to the server.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long      jnt_nr;
```
Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxSetErrorPolicy( CX_RETURN_ERRORS );
CxErrorLogPolicy( CX_LOG_TO_FILE, "error.log", CX_FALSE );

/* Open mechanism */
CxGetNamedNodeId( Server, "s100", &s100 );
mech = CxOpenMechanism( Server, s100, CX_CONTROL );

/* Move mechanism to perform desired task */
CxHomeMechanism( mech );
CxMoveSingleAxis( mech, 0, 100.0 );
CxMoveSingleAxis( mech, 1, 40.0 );
CxMoveSingleAxis( mech, 2, 15.0 );
CxMoveSingleAxis( mech, 3, 90.0 );
CxMoveSingleAxis( mech, 4, 175.0 );
CxWaitForEndOfMotion( mech );

/* you do not have to call CxCloseMechanism if CxRobpacExit is going to be called. CxRobpacExit calls CxCloseMechanism and CxCloseServer internally */
CxCloseMechanism( mech );
CxCloseServer( Server );
CxRobpacExit();

SEE ALSO
CxOpenMechanism, CxDetach, CxRobpacExit, CxSendMechanismErrorAction,
CxWaitForEndOfMotion, CxCloseServer
**CxCloseServer**

Closes a connection to an open CIMServer

**SYNOPSIS**

```
#include <code/robpac.h>
long CxCloseServer(CxServer server_to_close)
```

**ARGUMENTS**

- **server_to_close**: Server ID of the server to be closed; This is a CxServer id returned from a previous call to CxOpenServer.

**DESCRIPTION**

The CxCloseServer function terminates a connection with a CIMServer. The connection is severed by first closing all mechanisms opened within the calling process (see CxCloseMechanism). A message then informs the server that the connection to the calling process is terminating.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_CLOSE_SERVER_FAILED</td>
<td>Server could not be closed.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a server and a mechanism, then sets the error policy and error log policy to write errors to a file. Next, it homes the mechanism. (Note that in simulation mode, CxHomeMechanism has no effect). Then it moves different joints of the mechanism. Once all the moves are complete, it closes the mechanism and closes its connection to the server.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long      jnt_nr;
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxSetErrorPolicy( CX_RETURN_ERRORS );
    CxErrorLogPolicy( CX_LOG_TO_FILE, "error.log", CX_FALSE );

    /* Open mechanism */
```
CxGetNamedNodeId( Server, "s100", &s100 );
m mech = CxOpenMechanism( Server, s100, CX_CONTROL );

    /* Move mechanism to perform desired task */
    CxHomeMechanism( mech );
    CxMoveSingleAxis( mech, 0, 100.0 );
    CxMoveSingleAxis( mech, 1, 40.0 );
    CxMoveSingleAxis( mech, 2, 15.0 );
    CxMoveSingleAxis( mech, 3, 90.0 );
    CxMoveSingleAxis( mech, 4, 175.0 );
    CxWaitForEndOfMotion( mech );

    CxCloseServer( Server );
    CxRobpacExit();

SEE ALSO

CxCloseMechanism, CxOpenServer, CxRobpacExit, CxDetach
**CxDetach**

Closes connections to all open servers

**SYNOPSIS**

```
#include <code/robpac.h>
long CxDetach(void);
```

**ARGUMENTS**

This function does not require any arguments.

**DESCRIPTION**

This function closes connections with all CIMServer(s) opened using the CxOpenServer function. Internally, a list of all open servers is maintained when CxOpenServer and CxCloseServer are called. CxDetach is a convenience function, which traverses a list of open servers and closes them using CxCloseServer.

If an error occurs during one of the internal CxCloseServer calls, the function will continue to attempt to close any remaining open servers before returning an error condition.

This function is similar to CxRobpacExit, except that CxDetach does not exit the process upon completion.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_CLOSE_SERVER_FAILED</td>
<td>Server could not be closed.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following example sever the connection to several previously opened CIMServer's:

```

CxServer Server1, Server2;
Server1 = CxOpenServer( "Server1", CX_TCP_IP, 0 ) ;
Server2 = CxOpenServer( "Server2", CX_SMEM, 0 ) ;
/* Operations on all open servers */
CxDetach();
```
SEE ALSO

CxOpenServer, CxCloseServer
CxDisableAmps
Disables mechanism servo amplifiers

SYNOPSIS
#include <code/robpac.h>
long CxDisableAmps (CxMechanism mech)

ARGUMENTS
mech Mechanism ID. This is a CxMechanism ID returned by a call to CxOpen Mechanism.

DESCRIPTION
This function disables the specified mechanism’s servo amplifiers. Upon successful completion, the servo amplifiers will have been disabled. Before calling CxDisableAmps, the mechanism must be opened for CX_CONTROL.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX.MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX.MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>The specified mechanism must be open for CX_CONTROL when this function is called.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Mechanism ID entered is not valid.</td>
</tr>
</tbody>
</table>

EXAMPLE
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer myServer ;
    CxMechanism mech ;
    CxNodeId mech id ;
    long i, amps enabled, status ;

    /* Initialize connection to the server */
    myServer = CxOpenServer ( "MyServerName", CX_SMEM, 0 ) ;
    CxGetNamedNodeId ( myServer, "MyMech", &mech id ) ;
    mech = CxOpenMechanism ( myServer, mech id, CX_CONTROL ) ;

    /* Check to see if servo amplifiers are enabled, and if so disable them */
    CxGetControllerStatus ( mech, CX_AMPS_ENABLED, &status ) ;
if ( ( status & CX_AMPS_ENABLED ) )
{
    CxDisableAmps (mech) ;
}
printf ( "Amps disabled\n" ) ;

SEE ALSO
    CxEnableAmps, CxOpenMechanism
CxEnableAmps

Allows servo amplifiers to be enabled

SYNOPSIS

#include <code/robpac.h>

long CxEnableAmps (CxMechanism mech)

ARGUMENTS

mech    Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION

This function allows the servo amplifiers associated with a given mechanism to be enabled. When this
function returns successfully, the amplifiers have either been enabled or are in a mode in which they can be
enabled (using an external hardware switch, for example). The amplifier status when CxEnableAmps
returns is a function of the underlying controller type (see HARDWARE AND SYSTEM DEPENDENCIES
below). Before calling CxEnableAmps, the mechanism must be opened for CX_CONTROL.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>The specified mechanism must be open for CX_CONTROL when this function is called.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Mechanism ID entered is not valid.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following example allows the mechanism’s servo amplifiers to be enabled, then checks to ensure that
they were actually enabled before continuing.

    #include <stdio.h>
    #include <code/robpac.h>

    void main (void)
    {
        CxServer myServer ;
        CxMechanism mech ;
        CxNodeId mech_id ;
        long i, amps_enabled, status ;

        /* Initialize connection to the server */

        myServer = CxOpenServer ( "MyServerName", CX_SMEM, 0 ) ;
        CxGetNamedNodeId ( myServer, "MyMech", &mech_id ) ;
mech = CxOpenMechanism ( myServer, mech_id, CX_CONTROL ) ;
.
.
/*
Check to make sure servo amplifiers are enabled. Give user 5 chances to
correctly enable amps.
*/

amps_enabled = CX_FALSE ;
for ( i = 0 ; i < 5 ; i++ )
{
   CxGetControllerStatus ( mech, CX_AMPS_ENABLED, &status ) ;
   if ( !( status & CX_AMPS_ENABLED ) )
   {
      printf( "Enable servo amps and press <Enter> to continue\n" ) ;
      CxEnableAmps ( mech ) ;
      getchar ( ) ;
   }
   else {
      amps_enabled = CX_TRUE ;
      break ;
   }
}
if ( !amps_enabled )
{
   printf ( "Failure to enable servo amps ... Exiting.\n" ) ;
   CxRobpacExit ( ) ;
}

SEE ALSO

CxGetControllerStatus, CxSetControllerType, CxDisableAmps
**CxGetControlMode**

Finds the mode for which the mechanism is opened

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetControlMode (CxMechanism mech, long *mode)
```

**ARGUMENTS**

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **mode**: Mode for which the mechanism is opened. See DESCRIPTION below for possible modes.

**DESCRIPTION**

This function determines the mode for which the mechanism has been opened. The mode parameter is a bitmask, which can contain the following parameters:

- **CX_CONTROL**: The mechanism specified has been opened for **CX_CONTROL** by a process. Only one process may have a given mechanism open for **CX_CONTROL** at a time. If the mechanism is opened for **CX_CONTROL**, the process may edit or move that mechanism. If the mechanism is opened for **CX_MONITOR**, you cannot edit or move that mechanism but you can get all of the information about it. There is no limit on the number of processes and the number of times a mechanism may be opened for **CX_MONITOR** at one time. Note that this function does not report whether or not a mechanism has been opened in **CX_MONITOR** mode.

- **CX_KEY_NORMAL**: The mechanism specified has been opened for **CX_CONTROL** and **CX_KEY_NORMAL**. The meaning of this flag is dependent on the type of controller associated with the mechanism (see HARDWARE AND SYSTEM DEPENDENCIES for meaning of the **CX_KEY_NORMAL** flag).

- **CX_KEY_PENDANT**: The mechanism specified has been opened for **CX_CONTROL** and **CX_KEY_PENDANT**. The meaning of this flag is dependent on the type of controller associated with the mechanism (see HARDWARE AND SYSTEM DEPENDENCIES for meaning of the **CX_KEY_PENDANT** flag).

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (**CX_ERROR**) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>The client side mechanism is an invalid mechanism pointer.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

CxOpenMechanism
**CxGetControllerAttr**

Determines the attributes of the servo controller

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetControllerAttr (CxMechanism mech, long *attr_mask)
```

**ARGUMENTS**

- `mech` Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `attr_mask` This is a long value containing a bitmask of controller attributes.

**DESCRIPTION**

The function `CxGetControllerAttr` allows the user to determine the trajectory generation capabilities of the servo controller associated with an open mechanism. Upon successful completion, this function returns a bitmask in which the associated mask values are defined as follows:

- **CX_TRAJECTORY_MODE**: The servo controller accepts trajectory data.
- **CX_INVKIN_MODE**: The controller can generate inverse kinematic trajectories, such as linear and circular, given target frames.
- **CX_FWRDKIN_TIME_MODE**: The controller can generate forward kinematic trajectories given a target frame or joint values, and the time for which the motion should take place.
- **CX_FWRDKIN_RATE_MODE**: The controller can generate forward kinematic trajectories given a target frame or joint values and the joint rate at which the motion should take place.

The returned mask can be logically *and*-ed (&) with the above masks to determine if the servo controller has the specific capabilities.

If the servo controller’s trajectory generator supports **CX_TRAJECTORY_MODE**, then CIMControl can act as the trajectory generator, sending trajectory data to the servo controller at the specified servo rate (see `CxSetTrajectoryRate` in the Motion section of this manual).

If the servo controller supports **CX_INVKIN_MODE**, **CX_FWRDKIN_TIME_MODE**, or **CX_FWRDKIN_RATE_MODE**, the servo controller can accept a target frame or target joint values, and the servo controller will perform the appropriate type of trajectory generation. Otherwise, the servo controller must support **CX_TRAJECTORY_MODE**, and the trajectory generation will be performed by the CIMServer.

If the mechanism is in simulation mode, this API function has no meaning and will always return **CX_ALL_ATTR_MODE** which is defined as all of the modes OR-ed together.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>Error Codes</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_PROTOCOL_TYPE</td>
<td>A message transmission was attempted using an invalid messaging protocol.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following code determines if the servo controller associated with the mechanism supports trajectory data, then forces trajectory data to be transmitted to the servo card:

```
.CxServer Server;
CxMechanism mech;
long attr_mask;
CxNodeId Mech_node_id;
.
.
/* open mechanism */
mech = CxOpenMechanism (Server, Mech_node_id, CX_CONTROL) ;
if (!mech)
    return (CX_ERROR)
CxSetControllerType (Server, Mech_node_id, CX_PMAC_DRVR) ;
CxGetControllerAttr (mech, &attr_mask) ;
if (attr_mask & CX_TRAJECTORY_MODE)
    CxSetTrajectoryMode (mech,CX_CALCULATE_TRAJECTORY) ;
.
.
SEE ALSO

CxSetTrajectoryMode, CxSetControllerType
CxGetControllerFromMech

Gets controller ID for a given mechanism

SYNOPSIS

```c
#include <code/robpac.h>
CxController CxGetControllerFromMech (CxMechanism mech)
```

ARGUMENTS

- **Mech**  Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION

This function gets the controller ID of a mechanism controller associated with the given mechanism. This controller ID can then be used with the CxSendDeviceCommand function to send non-standard controller specific commands to the control interface for execution.

RETURN VALUES

This function returns a pointer to a valid CxController data structure if successful; otherwise, a CX_NULL pointer is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following example is used to determine a controller specific ID associated with a mechanism, and then get and set specific control parameters associated with the controller.

```c
#include <stdio.h>
#include <code/robpac.h>

char *server_name;
CxServer Server;
CxMechanism mech;
CxController Mcntrl;
long ivar, ival;

/* Establish connection with the server */

if ( GetDefaultServer ( &server_name ) == CX_ERROR )
    {
        printf ( "No Default Server Found\n" ) ;
        exit ( -1 ) ;
    }
```
Server = CxOpenServer ( server_name, CX_SMEM, 0 ) ;
CxGetNamedNodeId ( Server, "s100", &mech_node ) ;
CxSetControllerType ( Server, mech_node, CX_PMAC_DRVR ) ;
Mech = CxOpenMechanism ( Server, mech_node, CX_CONTROL ) ;
Mcntrl = CxGetControllerFromMech ( Mech ) ;

/* Set or get a controller specific control parameter */
do {
  fprintf ( stderr, "1. Get a controller I-variable\n" ) ;
  fprintf ( stderr, "2. Set a controller I-variable\n" ) ;
  fprintf ( stderr, "3. Exit\n" ) ;
  fflush ( stdio ) ;
  scanf ( "%ld", &choice ) ;
  switch ( choice )
  {
    case 1:
      fprintf ( stderr,"Enter the index of the desired I-var:");
      fflush ( stderr ) ;
      scanf ( "%ld", &ivar ) ;
      CxPmacGetIvar ( Mcntrl, ivar, &ival ) ;
      fprintf ( stderr, "I-var %ld has value %ld\n", ivar, ival ) ;
      break ;
    case 2:
      fprintf ( stderr,"Enter the index of the desired I-var:");
      fflush ( stderr ) ;
      scanf ( "%ld", &ivar ) ;
      fprintf ( stderr,"Enter the desired value:");
      scanf ( "%ld", &ival ) ;
      CxPmacSetIvar ( Mcntrl, ivar, ival ) ;
      fprintf ( stderr, "I-var %ld has been set to %ld\n", ivar, ival ) ;
      break ;
  }
} while ( choice != 3 ) ;
CxRopacExit ( ) ;

SEE ALSO

CxGetControllerFromSignal, CxSendDeviceCommand
CxGetControllerFromSignal

Gets the controller ID for the given signal

SYNOPSIS

```
#include <code/robpac.h>
CxController CxGetControllerFromSignal (CxServer Server, long signal)
```

ARGUMENTS

- **Server**: Server ID returned from call to CxOpenServer. This is a CxServer ID returned from a previous call to CxOpenServer.
- **signal**: Signal number

DESCRIPTION

This function gets the controller ID of a I/O controller associated with the specified signal. This controller ID can then be used with the CxSendDeviceCommand function to send non-standard controller specific commands to the control interface for execution. A number of utility functions specific to the CX_PMAC_DRVR have been built on the CxSendDeviceCommand function. (See the PMAC Specific Functions Section in the CODE API Reference – Vol. 3 manual for more information.)

RETURN VALUES

This function returns a pointer to a valid CxController data structure if successful; otherwise, a CX_NULL pointer is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is out of range of the signal table entries loaded in the CIMServer.</td>
</tr>
<tr>
<td>CX_NO_SIGNAL_FUNC_ROUTINE</td>
<td>The driver type of the given signal does not support this function.</td>
</tr>
<tr>
<td>CX_NO_MEMORY</td>
<td>Malloc failed trying to open a controller.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following example is used to determine a controller specific ID associated with a signal, and then gather data on specific signals.

```
#include <stdio.h>
#include <code/robpac.h>
.
.
char *server_name ;
CxServer Server ;
long gsignals[] = { DIN_1, DIN_2, DIN_3, DIN_4, DIN_5 } ;
char format[26], *buffer ;
```
double *data;
long data_size, act_cnt, count=0, buf_size, errno;
CxController Scntrl;

/* Establish connection with the server */

if ( GetDefaultServer ( &server_name ) == CX_ERROR )
{
    printf ( "No Default Server Found\n"
    );
    exit ( -1 );
}

Server = CxOpenServer ( server_name, CX_SMEM, 0 );
Scntrl = CxGetControllerFromSignal ( Server, gsignals[0] );

/* Test PMAC Data gathering */

printf ( "Testing gather_init\n" );
printf ( "Press <Enter> to start data gathering\n" );
fflush ( stdin );
getchar ( );
CxPmacGatherInit ( Scntrl, 5, gsignals, 100, 0, &count, 
&buf_size, format );
CxPmacGatherStart ( Scntrl, 0 );
printf ( "Gathering started for 5 seconds\n" );
sleep ( 5 );
CxPmacGatherStop ( Scntrl );
printf ( "Gathering stopped\n" );

buffer = ( char * ) malloc ( buf_size * sizeof ( char ) );
CxPmacGatherUpload ( Scntrl, 0, 0, &act_cnt, buf_size, buffer );

data = ( double * ) calloc ( 5 * act_cnt, sizeof ( double ) );
CxConvertGatherData ( buffer, format, data, &data_size, 
&errno );

printf ( "Number of data instances = %ld\n", act_cnt );
for ( j = 0 ; j < act_cnt ; j++ )
{
    printf ( "\nDATA INSTANCE %ld\n", j );
    for ( i = 0 ; i < 5 ; i++ )
        printf ( "Signal [%ld] = %lf\n", i, data[(j*5)+i] );
}

free ( ( char * ) data );
free ( ( char * ) buffer );

SEE ALSO

CxGetControllerFromMech, CxSendDeviceCommand, CxConvertGatherData
 CXGetControllerStatus
 Determines the operating status of a controller

 SYNOPSIS
 include <code/robpac.h>
 long CXGetControllerStatus (CxMechanism mech, long stat_mask, long *ret_stat_mask)

 ARGUMENTS
 mech Mechanism ID. This is a CxMechanism ID returned by a call to
 CxOpenMechanism.
 stat_mask This is a bitmask defining status attributes to be checked.
 ret_stat_mask This is a pointer to a long value containing a bitmask defining actual status of
 controller. The possible status bitmasks are listed below.

 DESCRIPTION
 This function determines the current operating status of a mechanism controller. The stat_mask argument
 defines which parameters are to be checked. If the corresponding bit is set in the ret_stat_mask argument,
 then that attribute is either operating satisfactorily or has been initialized; otherwise, it has not been
 initialized. The attribute masks, that can be checked are defined as follows:

 CX_CONTROLLER_INITIALIZED The controller has been initialized.
 CX_AMPS_ENABLED The servo amplifiers have been enabled.
 CX_MECHANISM_HOMED The mechanism has been homed.
 CX_TEACH_PENDANT_ACTIVE The controller’s teach pendant is active.
 CX_KEY_SWITCH_ACTIVE The controller’s key switch is active.

 A convenience mask (CX_CONTROLLER_STAT) allows you to check all of the defined attributes.

 If the mechanism is not in CX_RUNTIME mode, the ret_stat_mask returned is the same as the
 stat_mask passed in.

 RETURN VALUES
 This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

 ERRORS
 If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
 function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following example opens a mechanism, determines its current operating status, and ensures that everything is initialized before beginning the application:

```c
.CxServer Server;
CxMechanism mech;
CxNodeId mech_node_id;
long stat_mask;
.
mech = CxOpenMechanism (Server, mech_node_id, CX_CONTROL) ;
if (!mech)
    return (CX_ERROR) ;
CxGetControllerStatus (mech, CX_CONTROLLER_STAT, &stat_mask) ;
if ( (stat_mask & CX_CONTROLLER_STAT) != CX_CONTROLLER_STAT)
{
    if (! (stat_mask & CX_AMPS_ENABLED))
    {
        /* Enable Amps */
        CxEnableAmps(mech);
    }
    if (! (stat_mask & CX_MECHANISM_HOMED))
    {
        CxHomeMechanism (mech) ;
    }
}
```

SEE ALSO

CxHomeMechanism, CxEnableAmps
**CxGetControllerType**

Gets the type of controller associated with a mechanism

**SYNOPSIS**

```
#include <code/robpac.h>
long CxGetControllerType (CxServer Server, CxNodeId mech_id, long *type)
```

**ARGUMENTS**

- **Server**
  - Server ID; This is a CxServer ID returned from a previous call to CxOpenServer.
- **mech_id**
  - Node ID of mechanism associated with controller
- **type**
  - The controller type; see **DESCRIPTION** below for possible driver types

**DESCRIPTION**

This function returns the type of the current mechanism controller interface driver in the field pointed at by the type argument. Currently several driver types are supported by CODE which include the following:

- **CX_SOFTWARE**
  - Simulated testing of controller interaction
- **CX_AC28_DRVDR**
  - PAMUX driver interface
- **CX_DEVNET_DRVDR**
  - DeviceNet driver interface
- **CX_MEIDSP_DRVDR**
  - MEI-DSP device driver interface
- **CX_PCL722_DRVDR**
  - PCL722 device driver interface
- **CX_PMAC_DRVDR**
  - Delta Tau - PMAC device driver interface
- **CX_XMP_DRVDR**
  - MEI-XMP device driver interface.

These constants can be found in the header file `<code/cntr_const.h>`. You can add custom device driver interfaces by following the instructions in the *Customizing CODE* manual.

**RETURN VALUES**

This function returns **0** if successful; otherwise, **-1 (CX_ERROR)** is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_PROTOCOL_TYPE</td>
<td>A message transmission was attempted using an invalid messaging protocol.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following example makes sure that the controller type is set to **CX_PMAC_DRVDR** before the mechanism is opened:
CxServer Server;
CxMechanism mech;
CxNodeId mech_node_id;
long type;

CxGetControllerType (Server, mech_node_id, &type);
if (type != CX_PMAC_DRVR)
    CxSetControllerType (Server, mech_node_id, CX_PMAC_DRVR);

mech = CxOpenMechanism (Server, mech_node_id, CX_CONTROL);

SEE ALSO

CxOpenMechanism, CxSetControllerType
**CxGetCycleTime**

Gets the cycle time for the CIMServer to complete a move command

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetCycleTime(CxServer Server, double *time)
```

**ARGUMENTS**

- **Server**  
  Server ID.  
  This is a CxServer ID returned from a previous call to `CxOpenServer`.

- **time**  
  Time it takes to complete a command in seconds

**DESCRIPTION**

This function is used to determine the time elapsed in the execution of a process since the starting cycle time was initialized using the `CxSetCycleTime` function. If the server is in CX_ANIMATION mode, this time represents the theoretical time spent in processing motion commands. In CX_RUNTIME mode, this function returns the real process time.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the blend policy to CX_MOVE_WAIT so that the move functions will not return until the move is complete. Then it brings joints #0 and #1 to the 0.0 position. Next it resets the cycle time, makes a move, and gets the cycle time (the time taken to complete the first move). It then resets the time again and computes the time taken to complete the second move. It resets the cycle time once more, and makes two moves, one after the other. This cycle time is the time required to complete both moves.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    double    cycle_time;
    double    time_for_one_cycle;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);

    /* Open mechanism */
    CxGetNamedNodeID (Server, "s100", &s100);
```
mech = CxOpenMechanism (Server, s100, CX_CONTROL);
CxSetBlendPolicy (mech, CX_MOVE_WAIT);

/* Reset positions */
CxMoveSingleAxis (mech, 0, 0.0);
CxMoveSingleAxis (mech, 1, 0.0);

/* Measure time for each cycle */
CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 0, 100.0);
CxGetCycleTime (Server, &time_for_one_cycle);
printf ("Time for motion: %lf\n", time_for_one_cycle);

CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 1, 40.0);
CxGetCycleTime (Server, &time_for_one_cycle);
printf ("Time for motion: %lf\n", time_for_one_cycle);

/* Reset positions */
CxMoveSingleAxis (mech, 0, 0.0);
CxMoveSingleAxis (mech, 1, 0.0);

/* Measure combined cycle time */
CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 0, 100.0);
CxMoveSingleAxis (mech, 1, 40.0);
CxGetCycleTime (Server, &cycle_time);
printf ("Total Cycle time : %lf\n", cycle_time);

CxCloseMechanism (mech);
CxRobpacExit ();

SEE ALSO
CxSetCycleTime
**CxGetRoblineServerHost**

Gets the hostname of the CPU running CIMServer

**SYNOPSIS**

```c
#include <code/msgUtil.h>
char * CxGetRoblineServerHost (void)
```

**ARGUMENTS**

This function does not require any arguments.

**DESCRIPTION**

This function determines the current `robline_server_host` parameter, which is the hostname of the system where the target CIMServer is currently running. The `robline_server_host` parameter is used by the `CxOpenServer` function when attempting to establish a connection with a CIMServer using TCP/IP messaging protocol. By default, the server host is set to `localhost`, and can be set using the `CxSetRoblineServerHost` function. This function is not required when using CX_SMEM protocol.

**RETURN VALUES**

This function returns a pointer to a character string containing the current CIMServer host parameter, or a `CX_NULL` pointer if it is not set.

**EXAMPLE**

The following program gets and prints the security host name, the server host name and the server type.

**C Syntax**

```c
#include <stdio.h>
#include <string.h>
#include <code/robpac.h>
#include <code/msgUtil.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    char security_host_name[CX_MAX_HOSTNAME];
    char server_host_name[CX_MAX_HOSTNAME];

    long serv_type;

    /* Get default security host name */
    strcpy (security_host_name, CxGetSecurityHost());
    printf ('"Default security host is : %s \n", security_host_name);

    /* Get default host name */
    strcpy (server_host_name, CxGetRoblineServerHost());
    printf ('"Default server is : %s \n", server_host_name);
    Server = CxOpenServer( "Testing", CX_TCP_IP, 0 );

    /* Open mechanism */
    CxGetNamedNodeId (Server, "s100", &s100);
    mech = CxOpenMechanism (Server, s100, CX_CONTROL);
```
/* Get default server type */
CxGetServerType (mech, &serv_type);
printf ("Server type : %ld \n", serv_type);
.
.
}

SEE ALSO
CxOpenServer, CxSetRoblineServerHost
CxGetSecurityHost

Gets the hostname of the CPU running the security server

SYNOPSIS

#include <code/msgUtil.h>
char * CxGetSecurityHost (void)

ARGUMENTS

This function does not require any arguments.

DESCRIPTION

This function determines the hostname of the system on which the Cimetryx security server is running. This information can be set using the CxSetSecurityHost function (by default the security hostname is set to localhost). The security server is queried by some CODE API functions to get permission to execute and by the CxOpenServer API function when using TCP/IP messaging protocol. The security host parameter identifies the hostname to which the CODE application process must establish a connection in order to communicate with the security server. The name server must be started or running on this machine. This function is not needed when using CX_SMEM protocol.

RETURN VALUES

This function returns a pointer to a character string containing the current security host, or a CX_NULL pointer if the security host is not set.

EXAMPLE

The following program gets and prints the security host name, the server host name, the hitman path and the server type.

#include <stdio.h>
#include <string.h>
#include <code/robpac.h>
#include <code/msgUtil.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    char security_host_name[CX_MAX_HOSTNAME];
    char server_host_name[CX_MAX_HOSTNAME];
    long serv_type;

    /* Get default security host name */
    strcpy (security_host_name, CxGetSecurityHost());
    printf ("Default security host is : %s \n", security_host_name);

    /* Get default host name */
    strcpy (server_host_name, CxGetRoblineServerHost());
    printf ("Default server is : %s \n", server_host_name);

    Server = CxOpenServer ("Testing", CX_TCP_IP, 0);

    /* Open mechanism */
CxGetNamedNodeID (Server, "s100", &s100);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);

/* Get default server type */
CxGetServerType( mech, &serv_type);
printf ("Server type : %ld \n", serv_type);
.
.
}

SEE ALSO

CxSetSecurityHost
CxGetServerType

Gets the server type for the given mechanism

SYNOPSIS
#include <code/robpac.h>
long CxGetServerType (CxMechanism mech, long *type)

ARGUMENTS
mech Mechanism ID
type The mechanism’s server type (CX_ANIMATION or CX_RUNTIME).

DESCRIPTION
This function gets the server type for a given mechanism. A mechanism can be controlled in one of two
server modes: simulation (CX_ANIMATION) or runtime (CX_RUNTIME).

Under CIMulation, mechanisms are always controlled using the CX_ANIMATION server type. Under
CIMControl, the server type is typically set to CX_RUNTIME. In this case, all motion-related commands are
directed to a hardware interface library, which speaks to the mechanism’s servo controller. Alternatively
though, the server type can be set to CX_ANIMATION using CxSetServerType. With this server type,
the mechanism motions are simulated within CIMControl, which can be useful for detecting possible
collisions.

The CX_ANIMATION identifier is something of a misnomer, since animation is only performed under
CIMulation. It is a holdover from earlier CODE releases, in which the functionality of both CIMulation and
CIMControl were provided in a single server program. The name has been retained for historical purposes.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_SERV_TYPE</td>
<td>The mechanism corresponding to mech on the CIMServser</td>
</tr>
<tr>
<td></td>
<td>has an invalid server type.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program opens a server and a mechanism and then determines the server type.

#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
  CxServer Server;
CxMechanism mech;
CxNodeId   s100;
long       serv_type;

Server = CxOpenServer ("Testing", CX_SMEM, 0);

/* Open mechanism */
CxGetNamedNodeId (Server, "s100", &s100);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);

/* Get default server type */
CxGetServerType (mech, &serv_type);
if ( serv_type == CX_RUNTIME )
   printf ( "Mechanism s100 is in CX_RUNTIME mode\n" );
else
   printf ( "Mechanism s100 is in CX_ANIMATION mode\n" );

SEE ALSO

CxSetServerType
CxGetServerVersion

Gets the version of the server the client application is connected to

SYNOPSIS

#include <code\robpac.h>
long CxGetServerVersion(CxServer server, char* server_version);

ARGUMENTS

server The server the client application is connected to.
server_version A text string that contains the version number of the server.

DESCRIPTION

This API function returns the version of the CIMControl/CIMulation server, which the client application is connected to. The server_version is returned in the following format: <major release>.<minor release>.<patch release>.<build number>, e.g., 4.0.0.0025.

RETURN VALUES

This function returns CX_OK if successful; otherwise CX_ERROR is returned.

ERRORS

The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following example code segment shows the usage of CxGetServerVersion.

CxServer Server;
Char server_version[20];
...
CxGetServerVersion(Server, server_version);
**CxGetSimrate**

Gets the simulation rate (time step)

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetSimrate (CxServer Server, double *rate)
```

**ARGUMENTS**

- **Server**: Server ID
  - This is a `CxServer` ID returned from a previous call to `CxOpenServer`.
- **rate**: Size of the step the mechanism takes (in seconds)

**DESCRIPTION**

This function returns the current simulation time rate or simrate, for all the mechanisms in a graphical workcell. The higher this number, the faster the graphics will appear to move. Since the graphics redrawing also occurs at critical interaction points such as a mechanism pick or place of an object or at the end of a move, the redrawing of the screen does not always occur precisely at the simrate intervals. The trajectory generator actually performs at a higher frequency (smaller time intervals) than the simrate setting when in simulation mode so that the cycle time estimates remain accurate. The simrate only has meaning if the server used is CIMulation.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE FAILED</td>
<td>Error in receiving message.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`CxSetSimrate`
CxGetTrajectoryMode

Gets the trajectory mode flag

SYNOPSIS
#include <code/robpac.h>
long CxGetTrajectoryMode (CxMechanism mech, long *mode)

ARGUMENTS
mech   Mechanism ID
       This is a CxMechanism ID returned by a call to CxOpenMechanism.
mode   Trajectory mode. The possible modes are:
       CX_CALCULATE_TRAJECTORY and CX_END_POINT_MOVE.

DESCRIPTION
This function gets the current trajectory mode for trajectory generation. If the mode flag is
CX_CALCULATE_TRAJECTORY and the mechanism is in CX_RUNTIME mode, and the servo controller
supports CX_TRAJECTORY_MODE (see CxGetControllerAttr), then the trajectory information is
sent to the servo controller card at the specified servo rate.

If the mode flag is set to CX_END_POINT_MOVE and the controller supports the desired trajectory
generation type, then the target frame or joint values are sent to the servo controller. If the controller does not
support the trajectory generation type, then trajectory data is sent to the controller as long as
CX_TRAJECTORY_MODE is supported by the servo controller and the trajectory override flag is set (see
CxSetTrajectoryOverride).

If the server is being used in CX_ANIMATION mode, these functions do not affect the performance of the
simulation.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program gets and prints the values of trajectory mode setting and trajectory override flag
setting.

#include <stdio.h>
#include <code/robpac.h>
void main (void)
{
    CxServer    Server;
    CxMechanism mech;
CxNodeId s100;
long traj_mode, traj_override_flag;
long servo_supports;

Server = CxOpenServer("Testing", CX_SMEM, 0);

/* Open mechanism */
CxGetNamedNodeId(Server, "s100", &s100);
mec = CxOpenMechanism(Server, s100, CX_CONTROL);

CxGetTrajectoryMode(mech, &traj_mode);
printf("Trajectory mode is %ld \n", traj_mode);

CxGetTrajectoryOverride(mech, &traj_override_flag);
printf("Trajectory override flag is %ld \n", traj_override_flag);
}

SEE ALSO

CxSetTrajectoryMode, CxGetControllerAttr, CxGetTrajectoryOverride
CxGetTrajectoryOverride

Gets the current trajectory override setting

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetTrajectoryOverride (CxMechanism mech, long *trajectory_override)
```

ARGUMENTS

- **mech**
  - Mechanism ID.
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.

- **trajectory_override**
  - Flag containing 0 (trajectory override inactive) or 1 (trajectory override active)

DESCRIPTION

If the `trajectory_override` flag is set to `CX_TRUE` (default), a motion function is called with an interpolation type that the servo controller associated with the mechanism does not support. If the servo controller supports `CX_TRAJECTORY_MODE`, then CIMControl will override the `trajectory_mode` flag (see CxSetTrajectoryMode) and send the controller trajectory data.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program gets and prints the values of the trajectory mode setting and the trajectory override flag setting.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long       traj_mode, traj_override_flag;
    long       servo的支持;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);
```
/* Open mechanism */
CxGetNamedNodeID (Server, "s100", &s100);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);

CxGetTrajectoryMode (mech, &traj_mode);
printf("Trajectory mode is %ld \n", traj_mode);

CxGetTrajectoryOverride (mech, &traj_override_flag);
printf("Trajectory override flag is %ld \n", traj_override_flag);
.
.
}

SEE ALSO
CxSetTrajectoryOverride, CXGetTrajectoryMode
**CxHomeMechanism**

Homes a mechanism

**SYNOPSIS**

```
#include <code/robpac.h>
long CxHomeMechanism (CxMechanism mech)
```

**ARGUMENTS**

- **mech**
  - Mechanism ID.
  
  This is a CxMechanism ID returned by a call to CxOpenMechanism.

**DESCRIPTION**

This function homes the given mechanism. This API function only applies when the server in use is CIMControl. It has no effect when using CIMulation.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Cannot send message to the given server.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVED_FAILED</td>
<td>Cannot receive returned message from the given server.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a server and a mechanism, then sets the error policy and error log policy to write errors to a file. Next, it determines if the mechanism’s amplifiers need to be enabled or the mechanism needs to be homed. Then it moves different joints to perform a mechanism movement. Once all the motions are completed, it closes the mechanism and closes the connection to the server.

```c
#include <stdio.h>
#include <code/robpac.h>
#define MAX_RETRIES 5

void main (void)
{
  CxServer Server;
  CxMechanism mech;
  CxNodeId s100;
  long retries=0;

  Server = CxOpenServer ("Testing", CX_SMEM, 0);
  CxSetErrorPolicy (CX_RETURN_ERRORS);
  CxErrorLogPolicy (CX_LOG_TO_FILE, "error.log", CX_FALSE);

  /* Open mechanism */
```
CxGetNamedNodeId (Server, "s100", &s100);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);

/* Enable amps and Home the mechanism if required */
while ( retries++ < MAX_RETRIES )
{
    CxGetControllerStatus ( mech, (CX_AMPS_ENABLED | 
    CX_MECHANISM_HOMED),
                        &status ) ;
    if ( ! ( status & CX_AMPS_ENABLED ) )
    {
        printf ( "Enable amps and press <Enter> when ready\n" );
        CxEnableAmps ( mech ) ;
        fflush ( stdin ) ;
        getchar() ;
        continue ;
    }
    if ( !( status & CX_MECHANISM_HOMED ) )
    {
        CxHomeMechanism (mech);
        continue ;
    }
    break ;
}

/* Move mechanism to perform desired task */
CxMoveSingleAxis (mech, 0, 100.0);
CxMoveSingleAxis (mech, 1, 40.0);
CxMoveSingleAxis (mech, 2, 15.0);
CxMoveSingleAxis (mech, 3, 90.0);
CxMoveSingleAxis (mech, 4, 175.0);
CxWaitForEndOfMotion (mech);
CxRobpacExit ();

SEE ALSO

CxGetControllerStatus, CxEnableAmps
CxIsServerReady

Determines if a server is ready to receive client connections

SYNOPSIS

#include <code/robpac.h>
BOOL CxIsServerReady(char* ServerName, long protocol)

ARGUMENTS

ServerName  The name of the server the client wishes to connect to.
protocol    Type of connection protocol (CX_TCP_IP or CX_SMEM) to be used to connect to the server.

DESCRIPTION

This CODE API function can be used to determine if a CIMServer is ready to accept client connections.

RETURN VALUES

This function returns TRUE if the target server is ready to receive client connection requests. It returns FALSE if the server is not ready.

ERRORS

No error codes are generated with this function

EXAMPLE

This example makes ten attempts to determine if the default server is available for handling new connections, waiting one second between attempts.

```
#include <code/robpac.h>
#include<stdio.h>

void main()
{
    int i;
    char* serverName;
    if ( CxGetDefaultServer(&serverName) == CX_ERROR) {
        printf("No default server name available\n");
        exit(-1);
    }
    for (i = 0; i < 10; i++) {
        if (CxIsServerReady(serverName, CX_SMEM) == TRUE) {
            printf("%s is ready to accept connections\n", serverName);
            break;
        }
        Sleep(1000);
    }
    if (i == 10) {
        printf("Server %s is not ready to receive client connections.\n", serverName);
        exit(-1);
    }
    

}
```
SEE ALSO

CxWaitForServerReady, CxOpenServer
CxOpenMechanism

Opens a mechanism

SYNOPSIS

#include <code/robpac.h>
CxMechanism CxOpenMechanism (CxServer Server, CxNodeId mechanism, unsigned long mode)

ARGUMENTS

server  Server ID
  This is a CxServer ID returned from a previous call to CxOpenServer.
mechanism  Node ID of the mechanism node to be opened.
mode  This is a bitmask defining the mode for opening the mechanism.
  See DESCRIPTION below for possible open modes.

DESCRIPTION

This function establishes a connection with a mechanism defined in an open CIMServer. The mechanism can
be opened in either CX_MONITOR or CX_CONTROL modes, described as follows:

CX_MONITOR  Only CODE API functions, which monitor the state of the mechanism, can be called
  (e.g CxGetAxes, etc.). Any number of processes can open the mechanism as a
  CX_MONITOR mechanism.
CX_CONTROL  All mechanism-related functions can be called from that process. Only one process
  can open a mechanism in CX_CONTROL mode at a time.

Two other modes, CX_KEY_NORMAL and CX_KEY_PENDANT, are no longer supported.

It is important to note that under CIMControl, CxOpenMechanism also establishes a link with the physical
controller. The type of controller associated with the mechanism can either be saved with the workcell file
associated with the workcell, or set before calling CxOpenMechanism by calling the
CxSetControllerType function.

RETURN VALUES

This function returns a pointer to a CxMechanism data structure if successful; otherwise, CX_NULL is
returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function.
The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NO_MEMORY</td>
<td>The system has no available RAM for dynamic allocation.</td>
</tr>
<tr>
<td>CX_ROB_NOT_FOUND</td>
<td>The CxNodeId parameter does not correspond to a valid mechanism on the CIMServer.</td>
</tr>
<tr>
<td>CX_MECH_ALREADY_OPENED</td>
<td>The mechanism has been opened by another CxOpenMechanism call in CX_CONTROL mode.</td>
</tr>
<tr>
<td>CX_NO_OPEN_ROUTINE</td>
<td>No controller specific function exists for opening a device.</td>
</tr>
<tr>
<td>CX_NO_PMAC_DRIVER</td>
<td>The version of the CIMServer does not have the PMAC driver linked with it, and the controller_type is set to CX_PMAC_DRVR.</td>
</tr>
<tr>
<td>OAC_MECH_LOCKED</td>
<td>Mechanism is opened by another CIMServer.</td>
</tr>
<tr>
<td>CX_INVALID_OPEN_MODE</td>
<td>open_mode must be CX_MONITOR or CX_CONTROL.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following example demonstrates opening a CIMServer, getting the appropriate node IDs for the desired mechanisms, setting their controller types, and opening the mechanisms:

```c
#include <code/robpac.h>

void main (void)
{
    server peg;
    mechanism s100, gripper;
    CxNodeId s100_id, gripper_id;

    /* open server */
    peg = CxOpenServer ("Peg", CX_SMEM, 0) ;

    /* get all node ID to be used */
    CxGetNamedNodeId (peg, "gripper", &gripper_id) ;
    CxGetNamedNodeId ( peg, "s100", &s100_id) ;

    /* open mechanism */
    CxSetControllerType (peg, s100_id, CX_PMAC_DRVR) ;
    CxSetControllerType (peg, gripper_id, CX_PMAC_DRVR) ;

    /* Open mechanisms for CX_CONTROL, and require keyswitch to be in TEACH position for the s100 */
    s100 = CxOpenMechanism (peg, s100_id, CX_CONTROL|CX_KEY_PENDANT );
    gripper = CxOpenMechanism (peg, gripper_id,CX_CONTROL) ;
}
```
SEE ALSO

CxCloseMechanism, CxSetControllerType
**CxOpenServer**

Establishes a connection with a CIMServer

**SYNOPSIS**

```
#include <code/robpac.h>
CxServer CxOpenServer (char *server_name, long protocol, long tool_or_app)
```

**ARGUMENTS**

- server_name: The name of the server to be opened
- protocol: Type of messaging protocol (CX_SMEM or CX_TCP_IP, defined in `<code/basic_msg_const.h>`)  
- Tool_or_app: Tool or application identifier (1 for tool, 0 for application)

**DESCRIPTION**

This function establishes a connection with a CIMServer using a specified messaging protocol.

When the CIMServer is started, the logical name of the server is either specified on the command line, or read from the CimetrixInit file.

This option sets the logical name of the CIMServer. The server_name argument to the CxOpenServer function must match the server name of the running CIMServer.

When the CxOpenServer API function is called, it establishes a communication link between the calling process and the specified CIMServer. All subsequent CODE API function calls will communicate with the CIMServer by sending messages using the specified protocol. If the specified CIMServer is running on the same machine as the calling process, then the CX_SMEM or the CX_TCP_IP protocol can be used to communicate with the server. The CX_SMEM protocol is recommended in this situation because of its superior throughput (Note: The CX_SMEM protocol is the only protocol supported by the RTX version of CIMControl). If a non-RTX version of the CIMServer is running on a different machine, then only the CX_TCP_IP protocol can be used for communicating with that server.

When CxOpenServer is called with the CX_TCP_IP protocol, it first queries the Security Server to ensure that the specified server is running. The CxOpenServer function then establishes a connection directly with the CIMServer. In order to communicate with the Security Server, the CxSetSecurityHost function must be called before the call to CxOpenServer. (See CxSetSecurityHost). Otherwise, CxOpenServer searches the CimetrixInit file for the security host. If no security entry is found, localhost is used.

If using CX_TCP_IP protocol, CxSetRoblineServerHost must be called before calling CxOpenServer to set the hostname of the machine the CIMServer is running on. Otherwise, the CxOpenServer function assumes that the CIMServer is running on the same machine as the calling process.

The tool_or_app argument is used to distinguish an application from a tool (1 for tool, 0 for application). If a CODE application process connects to a server as a tool, any other process connected to the server (e.g. the CIMTools cell control panel) will not be notified when the process changes status (running, paused, stopped, etc.). The cell control panel in turn has no control over the tool process. On the other hand, if a CODE application process connects to a server as an application, the cell control panel will get notified regarding the application’s status, and be able to control the application process. When an application process is running, another process (tool or application) will not be allowed to clear the current workcell or open a new workcell file.
RETURN VALUES

This function returns a pointer to a valid CxServer data structure if successful; otherwise, CX_NULL is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NO_MEMORY</td>
<td>Machine out of memory.</td>
</tr>
<tr>
<td>CX_FIND_LISTENER_FAILED</td>
<td>Unable to establish link with CIMServer.</td>
</tr>
<tr>
<td>CX_NEW_ADDRESS_FAILED</td>
<td>Unable to allocate new message queue.</td>
</tr>
</tbody>
</table>

EXAMPLES

1. The following example illustrates opening a server using CX_SMEM protocol with the name server running on a machine called machineB.

```c
#include <code/robpac.h>

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId conveyor;

    /* open server */
    Server = CxOpenServer("HPdisk", CX_SMEM, 0);

    /* get node id’s */
    CxGetNamedNodeId (Server, "conveyor", &conveyor);

    /* open mechanism */
    mech = CxOpenMechanism Server, conveyor, CX_CONTROL);

    /* ... */
}
```

2. The following example demonstrates opening a CIMServer using CX_TCP_IP protocol.

```c
#include <code/robpac.h>

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId conveyor

    /* Set the code server host, and the security host */
CxSetRoblineServerHost ("machineA");
CxSetSecurityHost ("machineB");

/* open server */
Server = CxOpenServer ("HPdisk", CX_TCP_IP, 0);

/* get node id’s */
CxGetNamedNodeId (Server, "conveyor", &conveyor);

/* open mechanism */
mech = CxOpenMechanism (Server, conveyor, CX_CONTROL);

}

SEE ALSO

CxCloseServer, CxDetach, CxRobpacExit, CxSetRoblineServerHost, CxSetSecurityHost
**CxRobpacExit**

Cleans up the CODE API function interface and exits

**SYNOPSIS**

```c
#include <code/robpac.h>
void CxRobpacExit (void)
```

**ARGUMENTS**

This function does not require any arguments.

**DESCRIPTION**

This function severs connections with all open CIMServers and then exits the process. Internally, a list of all open Servers is maintained when `CxOpenServer` and `CxCloseServer` are called. The function `CxRobpacExit` is a convenience function, which calls `CxDetach` and then exits.

The function `CxDetach` traverses a list of open CIMServers and closes them using the `CxCloseServer` function. If an error occurs during one of the internal `CxCloseServer` calls, `CxDetach` will continue to attempt to close any remaining open CIMServers before returning an error condition.

This function is similar to `CxDetach`, except that `CxDetach` does not exit the process upon completion and `CxRobpacExit` does.

**RETURN VALUES**

This function does not return any values.

**ERRORS**

Since this function causes the process to exit, no errors are reported.

**WARNINGS**

Note that if `CxRobpacExit`, `CxDetach`, `CxCloseServer`, or `CxCloseMechanism` is not called before a process is allowed to exit, the CIMServer will not know that the process has exited, and will not allow other processes to establish connections with mechanisms opened using `CX_CONTROL` mode before the process exited. Note that the client process will trap and will inform the server that it has exited.

**EXAMPLE**

The following example uses `CxRobpacExit` to server connections with all open servers, and exit the process:

```c
void main (void)
{
    CxServer Server;
    CxMechanism mech ;
    CxNodeId mech_node_id ;
    .
    Server = CxOpenServer ( "MyServer", CX_SMEM, 0) ;
    CxGetNamedNodeId (Server, "s100", &mech_node_id) ;
    Mech = CxOpenMechanism (Server, mech_node_id, CX_CONTROL) ;
    .
    /* Process complete... exit */
    CxRobpacExit ( ) ;
}```
SEE ALSO

CxDetach, CxCloseServer, CxCloseMechanism
CxCxSendDeviceCommand
Sends a device specific command to the control interface

SYNOPSIS
#include <code/robpac.h>
long CxSendDeviceCommand (CxController cntrl, long queued, char
  *cmd_str, long size_of_cmd, char *ret_str, long *size_of_ret)

ARGUMENTS
   cntrl                     Controller ID
   queued                   Process command immediately or place in device controller’s processing queue
   cmd_str                   Controller specific command
   size_of_cmd               Number of bytes in cmd_str
   ret_str                   Response from device controller
   size_of_ret               Number of bytes in ret_str

DESCRIPTION
This functions allows commands specific to proprietary device controllers to be executed from a CODE
  application process. The controller ID obtained by calling either CxGetControllerFromMech for
  mechanism controllers or CxGetControllerFromSignal for I/O controllers.

The operation of the queued parameter is dependent on the implementation of the device controller. If the
  controller supports a command queue, and this flag is set to CX_TRUE, the command will be placed in the
  controllers command buffer, and will be executed after all previously received commands have completed. If
  the queued flag is CX_FALSE, the command will execute immediately.

The cmd_str parameter contains the desired command to be executed by the device controller. This
  command string cannot exceed CX_MAX_CMD_STR bytes (CX_MAX_CMD_STR is defined in
  <code/robconst.h>). Since it is possible for special characters to be embedded in this string, the length
  of the cmd_str must be specified in size_of_cmd parameter.

For many commands sent to a device controller, a response can be returned through the ret_str parameter.
  Since it is possible for special characters to be embedded in this string, the length of the ret_str is returned
  in the size_of_ret parameter.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
  function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVED_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>Error Codes</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CX_MAX_CMD_STR_LEN_EXCEEDED</td>
<td>The maximum message length for the cmd_str parameter has been exceeded. The maximum allowable length is defined by CX_MAX_CMD_STR.</td>
</tr>
<tr>
<td>CX_UNDEFINED_MECH_CMD_FUNC</td>
<td>A command has been sent to a mechanism controller, which the controller cannot interpret.</td>
</tr>
<tr>
<td>CX_UNDEFINED_SIG_CMD_FUNC</td>
<td>A command has been sent to an I/O controller, which the controller cannot interpret.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

The following example demonstrates sending a device specific command to a mechanism controller.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId mech_id;
    CxController Mcntrl;
    char send_command[MAX_CMD_LEN], response[MAX_CMD_LEN];
    long response_length;
    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
    CxGetNamedNodedId ( Server, "RT3200", &mech_id ) ;
    CxSetControllerType ( Server, mech_id, DARL_DRVR ) ;
    Mech = CxOpenMechanism ( Server, mech_id, CX_CONTROL ) ;
    Mcntrl = CxGetControllerFromMech ( Mech ) ;
    .
    .
    strcpy ( send_command, "DO HERE T1:DO PRINT T1" ) ;
    CxSendDeviceCommand ( Mcntrl, CX_FALSE, send_command,
            strlen ( send_command ), response, &response_length ) ;
    printf ( "Response = %s\n", response ) ;
    .
    .
}

**SEE ALSO**

CxGetControllerFromMech, CxGetControllerFromSignal
CxSetControllerType

Sets the type of controller associated with a mechanism

SYNOPSIS

```
#include <code/robpac.h>
long CxSetControllerType (CxServer Server, CxNodeId mech_id, long type)
```

ARGUMENTS

- **Server**
  
  Server ID
  
  This is a CxServer ID returned from a previous call to CxOpenServer.

- **mech_id**
  
  Node ID of mechanism associated with controller

- **type**
  
  Controller type (CXSOFTWARE, CX_PMAC_DRVR, etc., defined in <code/cntr_const.h> or user-defined controller type)

DESCRIPTION

This function allows the user to set the driver required to interface with a mechanism controller. The following device driver interfaces are supported by CODE:

- **CXSOFTWARE**
  
  Simulated testing of controller interaction

- **CX_AC28_DRVR**
  
  PAMUX device driver interface

- **CX_DEVNET_DRVR**
  
  Device Net device driver interface

- **CX_MEIDSP_DRVR**
  
  MEI-DSP device driver interface

- **CX_PCL722_DRVR**
  
  PCL722 device driver interface

- **CX_PMAC_DRVR**
  
  Delta Tau - PMAC device driver interface

- **CX_XMP_DRVR**
  
  MEI-XMP device driver interface

The controller type can be either saved in the CODE workcell file, so it is loaded when the CIMServer is started, or it can be set using the CxSetControllerType function. If set programmatically, the controller type should be set before calling CxOpenMechanism.

The user can add custom device driver interfaces by following the instructions in Chapter 4 of the Customizing CODE manual.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Node does not exist.</td>
</tr>
<tr>
<td>CX_ROB_NOT_FOUND</td>
<td>Given node is not valid. Node does not have ROBOT attribute.</td>
</tr>
<tr>
<td>Error Codes</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CX_CANT_SET_CONTROL_TYPE_RUNTIME</td>
<td>The controller type cannot be changed while the mechanism is open for CX_CONTROL or CX_MONITOR in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a server and a mechanism. It sets the controller type to CX_SOFTWARE.

**Note:** If the mechanism being controlled using the CX_RUNTIME server type and the controller type is set to CX_SOFTWARE then the user can simulate interaction with a mechanism controller.

```c
#include <stdio.h>
#include <code/robpac.h>
#include <code/cntr_const.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    CxSetErrorPolicy (CX_RETURN_ERRORS);

    /* Open mechanism */
    CxGetNamedNodeId (Server, "s100", &s100);
    CxSetControllerType (Server, s100, CX_SOFTWARE);
    mech = CxOpenMechanism (Server, s100, CX_CONTROL);

    CxMoveSingleAxis (mech, 0, 100.0);
    .

    CxRobpacExit ();
}
```

**SEE ALSO**

CxGetControllerType
**CxSetCycleTime**

Sets the cycle time for a CIMServer

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetCycleTime (CxServer Server, double time)
```

**ARGUMENTS**

- **Server**
  - Server ID
  - This is a CxServer ID returned from a previous call to CxOpenServer.
- **time**
  - Time to execute a move command (in seconds)

**DESCRIPTION**

This function allows the user to initialize the cycle time. Subsequent calls to CxGetCycleTime return time measured relative to this initial value. Under CIMulation or when CIMControl is used in CX_ANIMATION mode, the cycle time is a prediction of the time it would take to execute motion related commands, if the process were running in an on-line mode. When CIMControl is used in CX_RUNTIME mode, the cycle time reflects the real process time. Note that the cycle time is not reset after every move; hence, in order to reset it, the user must call this function.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Cycle time must be &gt;= 0.0.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the blend policy to CX_MOVE_WAIT so that the move functions will not return until each move is complete. Then it brings the joint #0 and #1 to 0.0 position. Next it resets the cycle time, makes a move, and gets the cycle time (the time taken to complete the first move). It resets the time again and computes the time taken to complete the second move. Then it resets the cycle time again, and makes two moves, one after the other. This cycle time is the time required to complete both moves.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    double    cycle_time;
    double    time_for_one_cycle;
```
Server = CxOpenServer ("Testing", CX_SMEM, 0);
/* Open mechanism */
CxGetNamedNodeId (Server, "s100", &s100);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);
CxSetBlendPolicy (mech, CX_MOVE_WAIT);
/* Reset positions */
CxMoveSingleAxis (mech, 0, 0.0);
CxMoveSingleAxis (mech, 1, 0.0);
/* Measure time for each cycle */
CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 0, 100.0);
CxGetCycleTime (Server, &time_for_one_cycle);
printf("Time for motion : %lf\n", time_for_one_cycle);
CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 1, 40.0);
CxGetCycleTime (Server, &time_for_one_cycle);
printf("Time for motion : %lf\n", time_for_one_cycle);
/* Reset positions */
CxMoveSingleAxis (mech, 0, 0.0);
CxMoveSingleAxis (mech, 1, 0.0);
/* Measure combined cycle time */
CxSetCycleTime (Server, 0.0);
CxMoveSingleAxis (mech, 0, 100.0);
CxMoveSingleAxis (mech, 1, 40.0);
CxGetCycleTime (Server, &cycle_time);
printf("Total Cycle time : %lf\n", cycle_time);
CxRobpacExit();

SEE ALSO
CxGetCycleTime
**CxSetRoblineServerHost**

Sets the CPU for connecting to CIMServer

**SYNOPSIS**

```c
#include <code/msgUtil.h>
long CxSetRoblineServerHost (char *hostname)
```

**ARGUMENTS**

- **hostname**  Hostname of the machine on which the CIMServer is running

**DESCRIPTION**

This function sets the current `robline_server_host` parameter, which is the hostname of the system on which the target CIMServer is currently running. The `robline_server_host` parameter is used by the `CxOpenServer` function when attempting to establish a connection with a CIMServer using TCP/IP messaging protocol. This function is not needed for `CX_SMEM` protocol. By default, the server host is set to `localhost`.

The hostname argument cannot exceed `CX_MAX_HOSTNAME` (defined in `<code/basic_msg_const.h>`) characters, including the terminating `CX_NULL` character; otherwise, an error will occur.

**RETURN VALUES**

This function returns `0` if successful; otherwise, `-1` (`CX_ERROR`) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_HOST_NAME_TOO_LONG</td>
<td>The hostname parameter is longer than <code>CX_MAX_HOSTNAME</code> characters.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the security host name and the server host name.

```c
#include <stdio.h>
#include <string.h>
#include <code/robpac.h>
#include <code/msgUtil.h>

void main (void)
{
    CxServer    Server
    char security_host_name[CX_MAX_HOSTNAME];
    char server_host_name[CX_MAX_HOSTNAME];
    /* Set security host name */
    strcpy (security_host_name, "moon");
    CxSetSecurityHost (security_host_name);

    /* Set server host name */
    strcpy (server_host_name, "localhost");
}
```
CxSetRoblineServerHost (server_host_name);

    Server = CxOpenServer ("Testing", CX_TCP_IP, 0);
    .
    .
}

SEE ALSO
CxGetRoblineServerHost, CxOpenServer
CxSetSecurityHost
Sets the CPU running Security

SYNOPSIS
#include <code/msgUtil.h>
long CxSetSecurityHost (char *hostname)

ARGUMENTS
hostname Hostname of the machine on which the Security Server is running

DESCRIPTION
The CxSetSecurityHost function sets the hostname of the system on which the Cimetrix security server 
is running.

In order to communicate with the security server, the CxSetSecurityHost function must be called 
before calling the CxOpenServer function (see CxOpenServer). Otherwise, CxOpenServer 
searches the CimetrixInit file for the security host. If no security entry is found, localhost is used.

This function has no effect when using CX_SMEM protocol.

The hostname argument cannot exceed CX_MAX_HOSTNAME (defined in <code/basic_msg_const.h>) characters, 
including the terminating CX_NULL character; otherwise, an error will occur.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber 
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_HOST_NAME_TOO_LONG</td>
<td>The hostname parameter is longer than CX_MAX_HOSTNAME characters.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program sets the security host name and the server host name. The server Testing is opened 
with TCP/IP protocol.

```
#include <code/robpac.h>
#include <code/msgUtil.h>

void main (void)
{
    CxServer Server

    /* Set security host name */
    CxSetSecurityHost ("moon");
    /* Set server host name */
    CxSetRoblineServerHost ("localhost");
```
Server = CxOpenServer ("Testing", CX_TCP_IP, 0);
   .
   .
}

The following program sets the security host name and the server Teaching is opened with Shared Memory (CX_SMEM) protocol.

#include <code/robpac.h>
#include <code/msgUtil.h>

void main (void)
{
   CxServer Server

   /* Set security host name */
   CxSetSecurityHost ("moon");

   Server = CxOpenServer ("Teaching", CX_SMEM, 0);
   .
   .
}

SEE ALSO

CxGetSecurityHost, CxOpenServer
**CxSetServerType**
Toggles between simulation and control for the given mechanism

**SYNOPSIS**
```
#include <code/robpac.h>
long CxSetServerType (CxMechanism mech, long type)
```

**ARGUMENTS**
- **mech** Mechanism ID of mechanism opened for **CX_CONTROL**.
  This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **type** Mechanism server type

**DESCRIPTION**
This function sets the server type for a given mechanism. A mechanism can be controlled in one of two server modes: simulation (**CX_ANIMATION**) or runtime (**CX_RUNTIME**).

Under CIMulation, mechanisms are always controlled using the **CX_ANIMATION** server type. Under CIMControl, the server type is typically set to **CX_RUNTIME**. In this case, all motion-related commands are directed to a hardware interface library which speaks to the mechanism's servo controller. Alternatively though, this API function can be used to specify the **CX_ANIMATION** type. With this type in effect, the mechanism motions are simulated within CIMControl, which can be useful for detecting possible collisions.

The "**CX_ANIMATION**" identifier is something of a misnomer, since animation is only performed under CIMulation. It is a holdover from earlier CODE releases, in which the functionality of both CIMulation and CIMControl were provided in a single server program. The name has been retained for historical purposes.

**RETURN VALUES**
This function returns **0** if successful; otherwise, **-1** (**CX_ERROR**) is returned and **1** (**CX_GOT_SIGNAL**) if:
- The process had previously registered an interrupt handler (such as CxInteruptOnValue) with the mechanism specified, and
- The signal interrupt occurs, and
- Previously queued motion was aborted using CxSendMechanismErrorAction in the signal handler.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_SERV_TYPE</td>
<td>Invalid server type for the given mechanism.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
The following program opens a server and a mechanism. It sets the simulation rate to 0.8, the server type to **CX_ANIMATION**, and the controller type to **CX_PMAC_DRVR**. It then allows a user to try a motion command in simulation before executing it on-line.
#include <stdio.h>
#include <code/robpac.h>
#include <code/cntr_const.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long axis;
    double value;
    char yesno;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    CxSetErrorPolicy (CX_RETURN_ERRORS);
    CxSetSimrate (Server, 0.8);

    /* Open mechanism */
    CxGetNamedNodeId (Server, "s100", &s100);
    CxSetControllerType (Server, &s100, CX_PMAC_DRVR);
    mech = CxOpenMechanism (Server, s100, CX_CONTROL);
    CxSetBlendPolicy (mech, CX_MOVE_WAIT);
    CxSetServerType (mech, CX_ANIMATION);

    while ( 1 )
    {
        printf ( "\nEnter axis to move:" ) ;
        fflush ( stdout ) ;
        scanf ( "%ld", &axis ) ;
        printf ( "\nEnter axis target value: " ) ;
        scanf ( "%lf", &value ) ;
        CxMoveSingleAxis ( mech, axis, value );
        printf("Would you like to actually perform the motion?(y/n): ");
        yesno = getchar ( ) ;
        if ( isupper ( yesno ) )
            yesno = ( char ) tolower ( ( int ) yesno ) ;
        if ( yesno == 'y' )
            CxSetServerType (mech, CX_RUNTIME);
            CxMoveSingleAxis ( mech, axis, value );
            CxSetServerType (mech, CX_ANIMATION);
    }
    printf("Would you like to do another move?(y/n ) : ");
    yesno = getchar ( ) ;
    if ( isupper ( yesno ) )
        yesno = ( char ) tolower ( ( int ) yesno ) ;
    if ( yesno == 'n' )
}
\{  
    break ;  
\}  

CxRobpacExit ();  

SEE ALSO  

CxGetServerType
**CxSetSimTrajectoryMode**

Sets the trajectory mode flag for simulation only

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetSimTrajectoryMode (CxServer Server, long mode)
```

**ARGUMENTS**

- **Server**
  - Server ID
  - This is a CxServer ID returned from a previous call to CxOpenServer.

- **mode**
  - Trajectory mode with possible values `CX_CALCULATE_TRAJECTORY`, or `CX_END_POINT_MOVE`.

**DESCRIPTION**

This function sets the trajectory mode for the current server. Valid options for mode are defined as follows:

- **CX_CALCULATE_TRAJECTORY**
  - This mode shows in simulation what the runtime mechanism would look like if trajectory information is sent to the servo controller card at the specified servo rate.

- **CX_END_POINT_MOVE**
  - This mode simulates the desired move without trajectory generation. The graphical model just moves from the beginning point of the move to the end point without simulating trajectory generation.

This function does not affect mechanisms, which have been opened under CIMControl, using the `CX_RUNTIME` server type.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The mode flag is neither <code>CX_CALCULATE_TRAJECTORY</code> nor <code>CX_END_POINT_MOVE</code>.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the value of simulation trajectory mode.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
```


{ 
    CxServer   Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    /* Open mechanism */
    CxGetNamedNodeId (Server, "s100", &s100);
    mech = CxOpenMechanism (Server, s100, CX_CONTROL);

    CxSetSimTrajectoryMode (Server, CX_CALCULATE_TRAJECTORY);
    .
    .
}

SEE ALSO
    CxSetTrajectoryMode
CxSetSimrate

Sets the simulation rate (time step) for all mechanisms in a workcell

SYNOPSIS

#include <code/robpac.h>
long CxSetSimrate (CxServer Server, double rate)

ARGUMENTS

Server  Server ID
This is a CxServer ID returned from a previous call to CxOpenServer.
rate  Size of the step the mechanism takes (> 0.0)

DESCRIPTION

This function sets the simulation rate for CIMulation. The higher the number, the faster the graphics will appear to move. The simulation rate applies to all mechanisms in a specified workcell. Because the graphics redrawing also occurs at critical interaction points such as a mechanism pick or place operation or at the end of a move, the redrawing of the screen does not always occur precisely at the simrate intervals. The trajectory generator actually performs at a higher frequency (smaller time intervals) than the simrate setting when in simulation mode so that the cycle time estimates remain accurate.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Specified rate is less than zero.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program opens a server and a mechanism. It sets the simulation rate to 0.8, and the controller type to CX_PMAC_DRVR.

#include <stdio.h>
#include <code/robpac.h>
#include <code/cntr_const.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long axis ;
    double value ;
    char yesno ;
    Server = CxOpenServer ("Testing", CX_SMEM, 0);
CxSetErrorPolicy (CX_RETURN_ERRORS);
CxSetSimrate (Server, 0.8);

/* Open mechanism */

CxGetNamedNodeId (Server, "s100", &s100);
CxSetControllerType (Server, &s100, CX_PMAC_DRVR);
mech = CxOpenMechanism (Server, s100, CX_CONTROL);
CxSetBlendPolicy (mech, CX_MOVE_WAIT);

while ( 1 )
{
    printf ( "\nEnter axis to move:" ) ;
    fflush ( stdout ) ;
    fflush ( stdin ) ;
    scanf ( "%d", &axis ) ;
    printf ( "\nEnter axis target value: " ) ;
    fflush ( stdout ) ;
    fflush ( stdin ) ;
    scanf ( "%lf", &value ) ;
    CxMoveSingleAxis ( mech, axis, value ) ;

    printf("Would you like to do another move?(y/n ): ");
    fflush ( stdout ) ;
    fflush ( stdin ) ;
    yesno = getchar () ;
    if ( isupper ( yesno ) )
    {
        yesno = ( char ) tolower ( ( int ) yesno ) ;
        if ( yesno == 'n' )
        {
            break ;
        }
    }
    CxRobpacExit () ;
}

SEE ALSO
   CxGetSimrate
**CxSetTrajectoryMode**

Sets trajectory mode flag

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetTrajectoryMode (CxMechanism mech, long trajectory_mode)
```

**ARGUMENTS**

- `mech` Mechanism ID
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `trajectory_mode` Trajectory mode with possible values `CX_CALCULATE_TRAJECTORY`, or `CX_END_POINT_MOVE`.

**DESCRIPTION**

This function sets the current trajectory mode for trajectory generation. Valid options for `trajectory_mode` are defined as follows:

- `CX_CALCULATE_TRAJECTORY` Trajectory information is sent to the servo controller card at the specified servo rate, provided that the servo controller supports `CX_TRAJECTORY_MODE` (see CxGetControllerAttr).

- `CX_END_POINT_MOVE` Target frame or joint values are sent to the servo controller, if the controller supports the desired trajectory generation type. If the controller does not support the trajectory generation type, then trajectory data is sent to the controller as long as `CX_TRAJECTORY_MODE` is supported by the servo controller and the trajectory override flag is set (see CxSetTrajectoryOverride).

This function only applies to mechanisms opened using CIMControl using the `CX_RUNTIME` server type. It does not affect the performance when the server type is `CX_ANIMATION`, such as when CIMulation is used.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The mode flag is neither 0 nor 1</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following program sets the value of trajectory mode.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;
    long attr;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;

    /* Open mechanism */
    CxGetNamedNodeId ( Server, "s100", &s100 ) ;
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;

    CxGetControllerAttr( mech, &attr ) ;
    if ( attr & CX_TRAJECTORY_MODE )
        CxSetTrajectoryMode ( mech, CX_CALCULATE_TRAJECTORY ) ;
.
.
}
```

SEE ALSO

CxSetTrajectoryOverride, CxGetTrajectoryMode, CxSetSimTrajectoryMode
CxSetTrajectoryOverride
Sets the current trajectory override setting

SYNOPSIS
#include <code/robpac.h>
long CxSetTrajectoryOverride (CxMechanism mech, 
long trajectory_override)

ARGUMENTS
mech Mechanism ID
This is a CxMechanism ID returned by a call to
CxOpenMechanism.

trajectory_override Trajectory override setting: 0 (disable trajectory override) or 1 (enable 
trajectory override)

DESCRIPTION
The trajectory override flag is used by CIMControl when it receives a motion command to determine how a 
trajectory will be generated, if the servo controller does not support the desired trajectory generation type.
If a motion function is called with an interpolation type that the mechanism’s controller does not support, and 
the servo controller supports CX_TRAJECTORY_MODE, CIMControl will override the 
trajectory_mode flag (see CxSetTrajectoryMode) and send the controller trajectory data. 
However, this occurs only if the trajectory_override flag is set to CX_TRUE (default). Otherwise an 
error (CX_TRAJECTORY_OVERRIDE_NOT_SET) is returned from the motion function.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber 
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE.Receive_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The flag argument is neither 0 nor 1.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following example disables the trajectory override flag to ensure that the operator uses the 
CxSetTrajectoryMode function when trajectory data is to be sent to the servo controller:

```

```
mech = CxOpenMechanism (Server, mech_node_id,CX_CONTROL) ;
.
.
CxSetTrajectoryOverride (mech, CX_FALSE) ;
if (CxMoveToNode (mech, targ_node_id, tcf_node_id ) == CX_ERROR)
{
    if ( CxGetErrorNumber (Server) == CX_TRAJECTORY_OVERRIDE_NOT_SET)
    {
        CxSetTrajectoryMode (mech,CX_CALCULATE_TRAJECTORY);
        CxMoveToNode (mech, targ_node_id, tcf_node_i) ;
    }
}

SEE ALSO

CxSetTrajectoryMode.CxGetControllerAttr.CxGetTrajectoryOverride
CxStopMechanism
Stops a mechanism

SYNOPSIS
#include <code/robpac.h>
long CxStopMechanism (CxMechanism mech)

ARGUMENTS
mech Mechanism ID
This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION
This function stops a moving mechanism. Once halted, the current motion and any movement pending in the
mechanism’s motion queue can be aborted or resumed (see CxSendMechanismErrorAction).

NOTE: CxSendMechanismErrorAction must be called after a call to CxStopMechanism before
the CIMServer will execute any other move calls associated with the stopped mechanism.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
</tbody>
</table>

EXAMPLES
The following example illustrates how the CxStopMechanism function can be used. In this example, a
mechanism is opened and its first axis is commanded to move to 30 degrees. At some point during the move
the user can stop the motion and be prompted as to what to do with the motion.

#include <code/robpac.h>
.
.
void main (void)
{
    CxServer myServer;
    CxMechanism mech;
    CxNodeId mech_id;
    long choice;
    
    myServer = CxOpenServer ("MyServer", CX_SMEM, 0);
    CxGetNamedNodeId (myServer, "MyMech", &mech_id);
    Mech = CxOpenMechanism (myServer, mech_id, CX_CONTROL);
    
    
}
CxSetBlendPolicy (Mech, CX_MOVE_TO);

CxMoveSingleAxis (Mech, 0, 30.0);
fprintf (stdout, "Press return when you want to stop\n");
getchar ();
CxStopMechanism (Mech);
fprintf (stdout, "What do you want to do now?\n");
fprintf (stdout, "1. Resume\n");
fprintf (stdout, "2. Abort\n");
fprintf (stdout, "3. Next\n");
fprintf (stdout, "4. Nothing\n");
scanf ("%ld", &choice);
switch (choice) {
  case 1:
    CxSendMechanismErrorAction (Mech, CX_MECH_RESUME);
    break;
  case 2:
    CxSendMechanismErrorAction (Mech, CX_MECH_ABORT);
    break;
  case 3:
    CxSendMechanismErrorAction (Mech, CX_MECH_NEXT);
    break;
}

SEE ALSO
CxSendMechanismErrorAction
CxWaitForServerReady

Waits for a server to be ready to receive client connections

SYNOPSIS

```c
#include <code/robpac.h>
#include <windows.h>

BOOL CxWaitForServerReady(char* ServerName, long protocol, 
DWORD timeout)
```

ARGUMENTS

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServerName</td>
<td>The name of the server the client wishes to connect to.</td>
</tr>
<tr>
<td>protocol</td>
<td>Type of connection protocol (CX_TCP_IP or CX_SMEM) to be used to connect to the server.</td>
</tr>
<tr>
<td>timeout</td>
<td>The number of milliseconds the function should wait before timing out. If the function should wait infinitely, use the value INFINITE.</td>
</tr>
</tbody>
</table>

DESCRIPTION

This CODE API function can be used to wait for a CIMServer to be ready to accept client connections.

RETURN VALUES

This function returns TRUE when the target server is ready to receive client connection requests. It returns FALSE if the timeout duration is exceeded.

ERRORS

No error codes are generated with this function.

EXAMPLE

This example waits ten seconds to determine if the default server is available.

```c
#include <code/robpac.h>
#include <stdio.h>
#include <windows.h>

void main()
{
    int i;
    char* serverName;

    if ( CxGetDefaultServer(&serverName) == CX_ERROR) {
        printf("No default server name available\n");
        exit(-1);
    }

    if (CxWaitForServerReady(serverName, CX_SMEM, 10000) == FALSE) {
        printf("Server %s is not ready to receive client connections.\n", serverName);
        exit(-1);
    }
    .
    .
}
SEE ALSO

CxIsServerReady, CxOpenServer
Motion Control

The functions in this library control the motion of mechanisms and other programmable devices in both simulation and on-line mode. These functions are relatively simple to use, and also very powerful. The “Motion” functions interact with the CODE API functions and the workcell database, using forward and inverse kinematics specified therein.

A mechanism can be opened either for control or for monitor purposes. A mechanism can only be controlled by a single process at a time, but may be monitored by several processes simultaneously. Some of the API functions in the library only work for mechanisms, which have been opened for control.
**CxClearStartSignal**

Disables the motion start signal of the given mechanism

**SYNOPSIS**
```
#include <code/robpac.h>
long CxClearStartSignal (CxMechanism mech)
```

**ARGUMENTS**
- **mech**
  - Mechanism ID of mechanism open for CX_CONTROL.
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.

**DESCRIPTION**
This function disables the motion start signal of the given mechanism. A motion start signal can be used to detect the beginning of a non-blended move for the given mechanism.

If a start signal is set using the non-blended CxSetStartSignal, the given signal will be set to the specified value whenever the non-blended motion begins. In order to ensure that the start signal is set when motion starts, the previous move command should have either a CX_MOVE_TO or CX_MOVE_WAIT blend policy. CxClearStartSignal disables the start signal.

Note that the start signal is not set during blended motions.

**RETURN VALUES**
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID. or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first initializes the software signal SW_SIG1 to 0. Next it sets up an interrupt handler for that signal in such a way that whenever the signal value reaches 1, that interrupt handler will be called. It designates signal SW_SIG1 as a start signal, so that whenever motion starts, that signal will be set to value 1. Then it moves an axis to a joint value of 100.0. As soon as motion starts the interrupt handler is invoked. The program waits until the motion is complete, then clears the start signal so that from the next move onward, signal SW_SIG1 does not get set to value 1, and the interrupt handler does not get called.

**NOTE:** It is assumed that SW_SIG1 and SW_SIG2 are defined in "sigTable.h."

```
#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h" /* signal table constant definitions */
```
void motion_started ( long signal, long value, CxMechanism mech);

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_WAIT );

    CxSetSignal ( Server, SW_SIG1, 0 );
    interrupt_on_value ( Server, SW_SIG1, 1, mech, motion_started );
    CxSetStartSignal ( mech, SW_SIG1, 1 );

    CxMoveSingleAxis ( mech, 0, 100.0 );

    CxClearStartSignal( mech );

    CxMoveSingleAxis ( mech, 0, 0.0 );

    CxRobpacExit ( );
}

void motion_started ( long signal, long value, CxMechanism mech )
{
    CxSendMechanismErrorAction ( mech, CX_MECH_RESUME );
    printf ( "Motion Started \n" );
    CxSetSignal ( Server, SW_SIG1, 0 );
}

SEE ALSO
    CxSetStartSignal
CxClearStopSignal

Disables the motion stop signal of the given mechanism

SYNOPSIS

#include <code/robpac.h>
long CxClearStopSignal (CxMechanism mech)

ARGUMENTS

mech  Mechanism ID of mechanism open for control.
       This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION

This function disables the motion stop signal of the given mechanism. A stop signal can be used to detect when a motion has successfully reached a target position.

If a stop signal is set (using the CxSetStopSignal API function), the given signal will be set to the specified value whenever a motion command reaches the target location until this function is called. The stop signal will not be triggered if the current move has a CX_MOVE_THRU blend policy or if the motion is aborted (see CxSendMechanismErrorAction). In order to ensure that the stop signal is set when a target is reached, the previous move command should have either a CX_MOVE_TO or CX_MOVE_WAIT blend policy.

Note that the stop signal is not set during blended motions.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program first initializes the software signal SW_SIG2 to 0. Next it sets up an interrupt handler for that signal in such a way that whenever the signal value reaches 1, that interrupt handler will be called. It designates signal SW_SIG2 as a stop signal so that whenever motion stops, that signal will be set to “1”. Then it moves an axis to a joint value of 100.0. As soon as motion completes, the interrupt handler gets invoked. After the first move, the program clears the stop_signal and from the next move on, signal SW_SIG2 does not get set to value 1 and the interrupt handler does not get called when motion stops.

NOTE: It is assumed that SW_SIG2 is defined in sigTable.h

#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h"
void motion_ended ( long signal, long value, CxMechanism mech );

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_WAIT );

    CxSetSignal ( Server, SW_SIG2, 0 );
    interrupt_on_value ( Server, SW_SIG2, 1, mech, motion_ended );
    CxSetStopSignal ( mech, SW_SIG2, 1 );

    CxMoveSingleAxis ( mech, 0, 100.0 );
    CxClearStopSignal ( mech );
    CxMoveSingleAxis ( mech, 0, 0.0 );
    CxRobpacExit();
}

void motion_ended ( long signal, long value, CxMechanism mech )
{
    CxSendMechanismErrorAction ( mech, CX_MECH_RESUME );
    printf(" Motion Ended \n");
    CxSetSignal ( Server, SW_SIG2, 0 );
}

SEE ALSO

CxSetStopSignal
**CxConstVelMoveAllAxes**

Disables the motion stop signal of the given mechanism

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxConstVelMoveAllAxes (CxMechanism mech, double *axes_vel)
```

**ARGUMENTS**

- **Mech**
  Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **axes_vel**
  This is an array of desired velocities for all joints of the mechanism. These values will be in degrees or radians per second for rotational joints. (See CxSetUnit in the Nodes, Frames & Attributes section of the CODE API Programmer’s Reference Manual – Volume 2).

**DESCRIPTION**

This function causes all of a robot’s independent joints to move in constant velocities forever until user calls this function again to specified zero velocities. Allowable motion interpolation type is CX_JOINT_INTERP, and motion blend policy is CX_MOVE_THRU. This API function only works in CX_RUNTIME mode.

**NOTE:** For purely serial robots, joint numbers begin with zero and increment along the robot linkage. All of the independent joints are numbered first, followed by all of the linearly dependent joints and, finally, all of the functionally dependent joints. For example, if a robot had six independent joints, two linearly dependent joints, and one functionally dependent joint, the independent joints would be numbered 0 through 5 in serial order, the linear joints would be 6 and 7, and the functional joint would be 8. Numbers are independent of any name you may assign in CODE.

**RETURN VALUES**

This function returns 0 if the move completes successfully; otherwise, -1 (CX_ERROR) is returned if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Trajectory generation error.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is in run-time mode, this function will cause the robot to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested with very low velocity first.

This function currently does not work in simulation mode.
EXAMPLE
The following code will move all six axes of a six degree-of-freedom robot in constant velocity 10 unit/s.

CxMechanism mech;
double axes_vel[CX_MAXJOINTS];
long i;

for (I=0; I<6, I++)
    axes_vel[I] = 10.0;
CxConstVelMoveAllAxes(mech, axes_vel);

SEE ALSO
CxConstVelMoveSingleAxis
**CxConstVelMoveSingleAxis**

Moves a single joint in specified constant velocity

**SYNOPSIS**

```
#include <code/robpac.h>
long CxConstVelMoveSingleAxes (CxMechanism mech, long axis, double axis_vel)
```

**ARGUMENTS**

- `mech`: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `axis`: The number of the joint to be moved.
- `axes_vel`: Desired constant velocity for the axis. A rotational joint’s velocity is defined in degrees or radians per second (see CxSetUnit in the Nodes, Frames & Attributes section of the CODE API Programmer’s Reference Manual – Volume 2 for more details). A translational joint’s velocity is defined in linear units per second (e.g. mm/s).

**DESCRIPTION**

This function causes a single axis to move in constant velocity. The specified axis must be independent. Allowable motion interpolation type is CX_JOINT_INTERP, and motion blend policy is CX_MOVE_THRU. This API function only works in CX_RUNTIME mode.

**NOTE:** For purely serial robots, joint numbers begin with zero and increment along the robot linkage. All of the independent joints are numbered first, followed by all of the linearly dependent joints and, finally, all of the functionally dependent joints. For example, if a robot had six independent joints, two linearly dependent joints, and one functionally dependent joint, the independent joints would be numbered 0 through 5 in serial order, the linear joints would be 6 and 7, and the functional joint would be 8. Numbers are independent of any name you may assign in CODE.

For nonserial robots, use the API function CxGetJntNumber to get the joint number of a named joint of a given mechanism.

This API works in CX_RUNTIME mode, but simulation ode of this API only works one axis a time. If user wants to command other axes move in simulation mode, he would have to configure the axes as different mechanism.

**RETURN VALUES**

This function returns 0 if the move completes successfully; otherwise, -1 (CX_ERROR) is returned if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_JOINT_NUMBER</td>
<td>Specified joint number is not valid.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Controller Management section of the CODE API Programmer’s</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Trajectory generation error.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is in run-time mode, this function will cause the robot to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested with very low velocity first.

In simulation mode, only one axis of a given mechanism can be moved at a time. For example, for an XY mechanism, the X-axis can be moved using CxConstVelMoveSingleAxis, the X-axis motion must be stopped before the Y-axis can be moved with the constant velocity move. In runtime, this restriction does not apply.

**EXAMPLE**

The following program opens a mechanism, sets the blend policy to CX_MOVE_THRU and issues commands to move joints. It moves ax #0 in velocity of 30 units/s, then changes to 60 units/s, and finally stops.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;

    Server = CxOpenServer("Testing", CX_SMEM,0);
    mech = CxOpenMechanism(Server, s100, CX_CONTROL);
    CxSetBlendPolicy(mech, CX_MOVE_THRU);

    CxConstVelMoveSingleAxis(mech, 0, 30.0);
    CxDelay(60,0);
    CxConstVelMoveSingleAxis(mech, 0, 0.0);
    CxWaitForEndOfMotion (mech);
    CxRobpacExit();
}
```

**SEE ALSO**

CxConstVelMoveAllAxes
CxDeleteConfig

Deletes a configuration associated with a mechanism

SYNOPSIS

```
#include <code/robpac.h>
long CxDeleteConfig (CxMechanism mech, char *name)
```

ARGUMENTS

- `mech`  Mechanism ID of mechanism open for CX_CONTROL; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `name`  Configuration name

DESCRIPTION

This function is used to delete a configuration associated with the specified mechanism. A configuration contains a set of joint values for a mechanism.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_CONFIG_NOT_EXIST</td>
<td>Named configuration does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program first creates a 3-D path, then saves the current configuration. Next, it moves the TCF along the path. It then moves the TCF back to the original configuration. Finally, it deletes the configuration.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100, path, tcf;
    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    CxGetNamedNodeId ( Server, "s100", &s100 );
    CxGetNamedNodeId ( Server, "path", &path );
    CxGetNamedNodeId ( Server, "tcf", &tcf );
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_WAIT);
```
/* Create a curve to be used for the tool movement */
CxAddCurveSeg ( Server, path, "", "pt1", "XYZ", 0.0, 0.0, 0.0,
0.0, 0.0, 0.0 );
CxAddCurveSeg ( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
0.0, 140.0, 500.0 );

CxTeachConfig ( mech, "orig_config", CX_TRUE, CX_ANIMATION );
CxMoveRelPath ( mech, path, tcf, "", "", "XYZ", 0.0, 0.0, 0.0,
5.0, 5.0, 5.0 );
CxMoveToConfig ( mech, "orig_config" );
CxDeleteConfig ( mech, "orig_config" );

CxRobpacExit ();
}

SEE ALSO
CxTeachConfig, CxListConfig
CxGetAccelType

Gets the current ramp acceleration type

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetAccelType (CxMechanism mech, long *type)
```

ARGUMENTS

- **mech**  
  Mechanism ID;  
  This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **type**  
  Mechanism acceleration type. See DESCRIPTION below for possible types.

DESCRIPTION

This function gets the current ramp acceleration type for a given mechanism. The two supported acceleration types are defined as follows:

- **CX_CONST_RAMP_TIME**  
  The time set for acceleration is a constant, thus the acceleration magnitude will be determined by the speed change required.
- **CX_CONST_RAMP_ACCEL**  
  A constant acceleration is used to accomplish speed change. The time will vary.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxSetAccelType
CxGetActualInterpType

Gets the current interpolation type for the moving mechanism

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetActualInterpType (CxMechanism mech, CxNodeId node, long *type)
```

**ARGUMENTS**

- **mech** Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **node** Mechanism node ID; This is the value returned by a call to CxGetNamedNodeId.
- **type** Interpolation type. See DESCRIPTION below for possible types.

**DESCRIPTION**

This function gets the current interpolation type of the specified mechanism (the specified node must have the ROBOT attribute). If the mechanism is not moving, then this value may not have any meaning. The returned interpolation type may differ from the value specified by CxSetInterpType since some type of motion override this setting (e.g., motion along a curve segment). The possible interpolation types are:

- **CX_JOINT_INTERP** The most efficient setting. Motors do not reverse direction or change speeds during a joint interpolated move. All joints arrive at the target settings simultaneously.
- **CX_LINEAR_INTERP** Moves TCF in a straight line from the current pose to a specified pose. This provides the most visually predictable path.
- **CX_LINEAR_JOG** Moves TCF in a straight line in a specified direction until an asynchronous event occurs (e.g., an asynchronous error or a call to CxStopMechanism).
- **CX_CIRCULAR_INTERP** This is the interpolation type used when moving along an arc or circular curve segment. The user cannot explicitly set the interpolation type to this value.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given node does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
<tr>
<td>CX_ATTRIBUTE_NOT_FOUND</td>
<td>Node does not have the ROBOT attribute.</td>
</tr>
</tbody>
</table>
The following program sets the interpolation type to `CX_JOINT_INTERP`. Next, it sets the joint speed to 40% of the maximum. Then it starts motion and goes into a loop, continuously monitoring the values of actual interpolation type and actual speed and prints them out.

```c
#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf;
    CxNodeId   t1, t2, t3;
    double    actual_speed;
    long      interp_type;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId ( Server, "S100", &MyMech );
    CxGetNamedNodeId ( Server "grip_tcf", &tcf );
    CxGetNamedNodeId ( Server "t1", &t1 );
    CxGetNamedNodeId ( Server "t2", &t2 );
    CxGetNamedNodeId ( Server "t3", &t3 );

    mech = CxOpenMechanism ( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_TO);
    CxSetInterpType ( mech, CX_JOINT_INTERP );
    CxSetJointSpeed ( mech, 0.4 );

    /* Move MyMech to perform a task */
    CxMoveToNode ( mech, t1, tcf );
    CxMoveToNode ( mech, t2, tcf );
    CxMoveToNode ( mech, t3, tcf );

    /* Continuously observe MyMech parameters */
    while (1)
    {
        CxGetActualInterpType ( mech, MyMech, &interp_type );
        CxGetActualJointSpeed ( mech, MyMech, &actual_speed );
        printf(" Current interpolation type is : %ld \n", interp_type );
        printf(" Current tcf node speed is     : %lf \n", actual_speed );
    }
}

SEE ALSO

CxGetInterpType, CxSetInterpType
```
**CxGetActualInvkinSoln**

Returns the kinematics solution used in the most recent or current move.

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetActualInvkinSoln (CxMechanism mech, CxNodeId node, long *soln)
```

**ARGUMENTS**

- `mech` Mechanism ID;
  
  This is a CxMechanism ID returned by a call to CxOpenMechanism.

- `node` Mechanism node ID;
  
  This is the value returned by a call to CxGetNamedNodeId.

- `soln` Inverse kinematics solution used.

**DESCRIPTION**

If the inverse kinematics solution setting is `CX_ALL_SOLN`, this function returns the number of the inverse kinematics solution used in the current or most recent move. This number may vary because of joint limitations and other constraints. If the invkin solution setting is not `CX_ALL_SOLN`, this function will return the specific solution number set for the inverse kinematics solution (see CxSetInvkinSoln and CxGetInvkinSoln). The specified node must have the ROBOT attribute.

**RETURN VALUES**

This function returns `0` if successful; otherwise, `-1` (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given mechanism does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first establishes a connection with the server and gets node ids for various nodes. Next it sets the inverse kinematic solution type to `CX_ALL_SOLN`. Then it moves the TCF of the mechanism to various positions. After every move it gets the actual solution number used and prints it out.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
```
CxMechanism mech;
CxNodeId   MyMech, tcf, t1, t2, t3;
long       soln_used;

Server = CxOpenServer ("Testing", CX_SMEM, 0);
CxGetNamedNodeId (Server, "MyMech", &MyMech);
CxGetNamedNodeId (Server, "tcf", &tcf);
CxGetNamedNodeId (Server, "t1", &t1);
CxGetNamedNodeId (Server, "t2", &t2);
CxGetNamedNodeId (Server, "t3", &t3);

mech = CxOpenMechanism (Server, MyMech, CX_CONTROL);
CxSetInvkinSoln (mech, CX_ALL_SOLN);
CxSetInterpType (mech, CX_LINEAR_INTERP);
CxSetBlendPolicy (mech, CX_MOVE_WAIT);

CxMoveToNode (mech, t1, tcf);
CxGetActualInvkinSoln (mech, MyMech, &soln_used);
printf(" Actual soln # used : %ld \n", soln_used);
CxMoveToNode (mech, t2, tcf);
CxGetActualInvkinSoln (mech, MyMech, &soln_used);
printf(" Actual soln # used : %ld \n", soln_used);
CxMoveToNode (mech, t3, tcf);
CxGetActualInvkinSoln (mech, MyMech, &soln_used);
printf(" Actual soln # used : %ld \n", soln_used);

CxRobpacExit();

SEE ALSO
  CxGetInvkinSoln, CxSetInvkinSoln
CxGetActualJointSpeed

Gets the current percent joint speed for a joint-interpolated move

SYNOPSIS

#include <code/robpac.h>
long CxGetActualJointSpeed(CxMechanism mech, CxNodeId node, double *speed)

ARGUMENTS

mech    Mechanism ID;
        This is a CxMechanism ID returned by a call to CxOpenMechanism.
node    Mechanism node ID;
        This is the value returned by a call to CxGetNamedNodeId.
speed   Percent joint speed in the range 0.0 to 1.0 where 0.5 would represent 50% of the mechanism’s maximum speed

DESCRIPTION

This function gets the percent joint speed used in the current move when the move’s interpolation type is CX_JOINT_INTERP. In simulation mode, this value will always be the same as the setting returned using the CxGetJointSpeed function. The specified node must have the ROBOT attribute.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given mechanism does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program sets the interpolation type to CX_JOINT_INTERP. Next it sets the joint speed to 40% of the maximum. It starts motion and executes in a loop, continuously printing out the actual interpolation type and joint speed.

```c
#include <stdio.h>
#include <code/robpac.h>
void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId    MyMech, tcf;
    CxNodeId    t1, t2, t3;
```
double actual_speed;
long interp_type;

Server = CxOpenServer ("Testing", CX_SMEM, 0);
.
.
m = CxOpenMechanism (Server, MyMech, CX_CONTROL);
CxSetBlendPolicy (mech, CX_MOVE_TO);

CxSetInterpType (mech, CX_JOINT_INTERP);
CxSetJointSpeed (mech, 0.4);

/* Move MyMech to perform a task */
CxMoveToNode (mech, t1, tcf);
CxMoveToNode (mech, t2, tcf);
CxMoveToNode (mech, t3, tcf);

/* Continuously observe MyMech parameters */
while (1)
{
    CxGetActualInterpType (mech, MyMech, &interp_type);
    CxGetActualJointSpeed (mech, MyMech, &actual_speed);
    printf("Current interpolation type is : %d \n", interp_type);
    printf("Current tcf node speed is : %lf \n", actual_speed);

    CxGetActualInterpType (mech, MyMech, &interp_type);
    CxGetActualJointSpeed (mech, MyMech, &actual_speed);
    printf("Current interpolation type is : %d \n", interp_type);
    printf("Current tcf node speed is : %lf \n", actual_speed);
}

SEE ALSO
CxGetJointSpeed, CxSetJointSpeed
**CxGetAxes**

Gets a mechanism’s current axis values

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetAxes (CxMechanism mech, double *axes_values)
```

**ARGUMENTS**

- **mech**
  
  Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.

- **axes_values**
  
  Current axes’ values (if rotational joint, units are in degrees unless the unit is set to CX_USE_RADIAN. See CxSetUnit in the Nodes, Frames & Attributes section of the CODE API Programmer’s Reference Manual - Volume 2 for more details.)

**DESCRIPTION**

This function gets the current absolute axes values for the given mechanism.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MACHINE_OUT_OF_MEMORY</td>
<td>Memory allocation error.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following segment gets the current axes values for the mechanism s100:

```
long i;
CxMechanism s100;
double jv[6];
.
.
CxGetAxes ( s100, jv );
for ( i=0; i<6; i++ )
    printf( "joint[%d] = %lf\n", i, jv[i] );
```

**SEE ALSO**

CxGetUnit, CxSetUnit
**CxGetBlendPolicy**

Gets the mechanism’s current blend policy

**SYNOPSIS**

```c
#include <code/robpac.h>
l.long CxGetBlendPolicy(CxMechanism mech, long *policy)
```

**ARGUMENTS**

- **mech**: Mechanism ID;
  This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **policy**: Policy which is used to determine whether blending is used and how it is performed.

**DESCRIPTION**

This function is used to determine the mechanism’s current blend policy. The blend policy defines whether or not a motion comes to a complete stop before initiating the next move in the Server’s motion queue. It also defines if a motion related function call will return immediately after the move is placed in the motion queue or wait until the move has finished processing. Possible values for the blend policy are defined as follows:

- **CX_MOVE_TO**: This policy will load a move into a move queue and return immediately (it will not wait for the motion to complete). A motion command with this blend policy will come to a complete stop before initiating the next move in the queue.

- **CX_MOVE_THRU**: This policy will load a move into a move queue and return immediately (it will not wait for the move to complete). Motion commands with this blend policy will not come to a complete stop if there is another move following in the queue. Instead, the next move in the queue will be blended with the current move.

- **CX_MOVE_WAIT**: Motion commands with this blend policy will load a move into the queue and will wait until all previously queued moves and the current commanded move are completed or aborted before returning. Motions with this blend policy will also come to a complete stop before returning. The default blend policy is **CX_MOVE_WAIT**.

- **CX_MOVE_TO_TANGENT**: This blend policy should be applied to paths of the type `curveseg` (linked list of curvilinear segments). Tangent segments will be transitioned exactly without blending, maintaining the desired speed, while non-tangent segments will have their blend policy set to **CX_MOVE_TO**.

- **CX_MOVE_THRU_TANGENT**: This blend policy should be applied to paths of the type `curveseg` (linked list of curvilinear segments). Tangent segments will be transitioned exactly without blending, maintaining the desired speed, while non-tangent segments will have their blend policy set to **CX_MOVE_THRU**.

See the CODE Applications Programming manual for a description of blended and non-blended moves.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first sets the blend policy to `CX_MOVE_WAIT` and then issues commands to move joints. Then it gets the blend policy for the mechanism and prints it out. The print message appears only after both moves are complete. Then it sets the blend policy to `CX_MOVE_TO` and does the same thing. This time the message appears even though the motion is not complete.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;
    long policy;

    Server = CxOpenServer("Testing", CX_SMEM, 0);

    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_WAIT);
    CxMoveSingleAxis ( mech, 0, 100.0 );
    CxMoveSingleAxis ( mech, 1, 120.0 );
    CxGetBlendPolicy ( mech, &policy );
    printf(" Current policy is : %ld \n", policy );

    CxSetBlendPolicy ( mech, CX_MOVE_TO );
    CxMoveSingleAxis ( mech, 0, 0.0 );
    CxMoveSingleAxis ( mech, 1, 0.0 );
    CxGetBlendPolicy ( mech, &policy );
    printf(" Current policy is : %ld \n", policy );
    CxWaitForEndOfMotion ( mech );

    CxRobpacExit();
}
```

**HARDWARE AND SYSTEM DEPENDENCIES**

Move types and blending capabilities depend on controller hardware and software. It is not safe to assume that all controllers support these motion blending capabilities. With CODE 4.0, CIMControl supports blended moves on the PMAC and MEI DSP cards, but not on the MEI XMP card.

**SEE ALSO**

CxSetBlendPolicy
CxGetBlendType

Gets the mechanism’s current blend type

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetBlendType(CxMechanism mech, long *type)
```

ARGUMENTS

- `mech` Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `type` Type of blending currently set. See DESCRIPTION below for possible blend types.

DESCRIPTION

This function is used to get the specified mechanism’s current blend type. The blend type is specified as one of two types:

<table>
<thead>
<tr>
<th>Blend Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_DECEL_BLEND</td>
<td>This is the normal default blend type which starts blending two moves when the first move begins its deceleration period to zero speed. The second move begins its accel period at the exact time the first begins its decel period and both moves are integrated together to maintain the speed near the desired speed setting.</td>
</tr>
<tr>
<td>CX_PERCENT_SPEED_BLEND</td>
<td>This blend type uses the decel blend type except the accel period of the second blended move is delayed until the speed of the first move is reduced to some percentage of the speed set for the move. Another API function, CxSetPercentBlendSpeed() is required to set the percent of the set speed at which blending is to occur; otherwise, the default percentage of 100% is used.</td>
</tr>
</tbody>
</table>

**NOTE:** The user must enter a number between 0.0 and 1.0 rather than an actual percentage. The default value is 1.0 (see the CODE Applications Programming manual for a description of blended and non-blended moves.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following program gets the current blend type, and then uses the value returned to change the blend type.

```c
#include <stdio.h>
#include <code/robpac.h>

void main( int argc, char **argv )
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;
    long type;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
    .
    .
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;

    CxGetBlendType ( mech, &type );

    if(type == CX_PERCENT_SPEED_BLEND)
        CxSetBlendType ( mech, CX_DECEL_BLEND);
    .
    .
    CxRobpacExit();
}
```

WARNINGS

The user must be sure that the motion card is capable of blending in order to take advantage of this function. Currently, CIMControl only supports blending on the PMAC and MEI DSP motion cards.

HARDWARE AND SYSTEM DEPENDENCIES

Move types and blending capabilities depend on the controller hardware and software. It is not safe to assume that all controllers support these move capabilities. With CODE 4.0, CIMControl supports blended moves on the PMAC and MEI DSP cards, but not on the MEI XMP card.

SEE ALSO

CxSetBlendPolicy, CxSetPercentBlendSpeed, CxSetBlendType
**CxGetConfigName**

Gets the name of a configuration for a given mechanism and index

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetConfigName(CxMechanism mech, long index, char *name)
```

**ARGUMENTS**

- `mech` Mechanism ID
  - This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `index` Index to configuration array for the given mechanism
- `name` Name of the configuration

**DESCRIPTION**

This function gets the name of the configuration with a specified index. When a configuration is taught, the name of the configuration and the corresponding joint values are saved in a configuration array. This function allows the user to get the name of any configuration stored in the configuration array associated with the mechanism.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>index is outside the configuration array</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following process gets the total number of configurations taught for mechanism s100, and then lists the configuration names to stderr.

```c
#include <stdio.h>
#include <code/robpac.h>

main ( void )
{
    CxServer peg;
    CxMechanism s100;
    long i, num;
    char name[CX_MAXNAME];
    CxNodeId s100_id;
```
/* open server */
    peg = CxOpenServer("Peg", CX_SMEM, 0);
    CxGetNamedNodeId(peg, "s100", &s100_id);
    s100 = CxOpenMechanism(peg, s100_id, CX_CONTROL);
    CxSetErrorPolicy(CX_RETURN_ERRORS);
    CxGetNumConfig(s100, &num);
    for (i=0; i<num; i++)
    {
        CxGetConfigName(s100, i, name);
        fprintf(stderr,"configuration %ld: %s\n", i+1, name);
    }
    CxRobpacExit();

SEE ALSO
    CxGetNumConfig, CxListConfig, CxMoveToConfig, CxTeachConfig, CxDeleteConfig
CxGetInnerTcfOffset

Gets attained TCF target offsets relative to desired target

SYNOPSIS

#include <code/robpac.h>
long CxGetInnerTcfOffset(CxMechanism mech, CxNodeId node, char axes[4],
CxVector angles, CxVector vec)

ARGUMENTS

mech    Mechanism ID
        This is a CxMechanismID returned by a call to CxOpenMechanism.

node    The TCF node. This value can be obtained by calling CxGetNamedNodeId.

axes    Returns the principal axes about which the frame is rotated, and the order of the rotation (e.g.
        XYZ, ZXZ, etc.).

angles  Returns the relative rotation angles in the same order as the axes were returned. The returned
        angles are in degrees unless the units are set to radians. See example and CxSetUnit in the
        Nodes, Frames & Attributes section of the CODE API Programmer’s Reference Manual –
        Volume 2 for more details.

vec     Returns the relative position vector.

NOTE: Type CxVector is defined in <code/matx_def.h> as follows:

typedef double CxVector[3];

DESCRIPTION

The inverse kinematics routines for a tool control frame (TCF) are meant to determine joint values that will
place the TCF at the target frame. Often, this is not possible when the TCF is attached to an interior joint. In
many such cases, the inverse kinematics solution will successfully place the TCF origin at the target origin,
but not match the target orientation.

This API function allows the user to determine the offsets of the attained target from the desired target,
measured in the desired target frame. This offset is determined relative to the target that is specified by the
move command. For example, if a CxMoveRelNode() move command is used to specify an offset target,
then the inner TCF offsets will be reported relative to this offset target frame rather than the node frame itself.

WARNINGS

This function should only be used for inner TCF’s when the blend policy is set to CX_MOVE_WAIT. Non-
zero values will only be returned when this API function is called immediately after the move and offsets
actually occur. This function can also be used when the blend policy is CX_MOVE_TO, provided that
CxWaitForEndOfMotion() immediately follows the move for which the offsets are desired. If these
rules are not followed, then the offsets returned will not be valid.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrNumber
function. The possible error codes are defined in the following table:
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>index is outside the configuration array</td>
</tr>
</tbody>
</table>
| CX_INVALID_MECHANISM                   | Invalid mechanism ID, or mechanism does not exist.

**EXAMPLE**

The following example code will get the attained TCF offsets relative to the target, which is the node with node_id_part. It returns the rotation axes (axes, in the order the rotations take place), the rotation angles (ang, in degrees, about those axes in the same order), and the relative position vector (vec).

```c
CxMechanism mech;
CxNodeId part, tcf;
char axes[4];
CxVector ang, vec;

CxSetBlendPolicy(mech, CX_MOVE_WAIT);

CxMoveToNode(mech, part, tcf);
CxGetInnerTcfOffset(mech, tcf, axes, ang, vec);

If you want CxGetInnerTcfOffset to return angles in radians, simply do following:

```c
CxSetUnit(1.0, CX_USE_RADIAN);
CxMoveToNode(mech, part, tcf);
CxGetInnerTcfOffset(mech, tcf, axes, ang, vec);
```

Note that the unit specified via CxSetUnit affects all successive API functions, except those in the Matrix library. If necessary, the units can always be set back to the preferred unit.

**SEE ALSO**

CxMoveToNode, CxMoveRelNode
CxGetInterpType

Gets the current motion interpolation type to be used in target moves

SYNOPSIS

#include <code/robpac.h>
long CxGetInterpType(CxMechanism mech, long *interp_type)

ARGUMENTS

mech Mechanism ID;
This is a CxMechanism ID returned by a call to CxOpenMechanism.

interp_type Current motion interpolation type.
See DESCRIPTION below for possible interpolation types.

DESCRIPTION

This function gets the mechanism’s current interpolation type, which can be one of the following:

CX_JOINT_INTERP The most efficient setting. Motors do not reverse direction or change speeds during a joint interpolated move. All joints arrive at the target settings simultaneously.

CX_LINEAR_INTERP Moves TCF in a straight line from the current pose to a specified pose. This provides the most visually predictable path.

CX_LINEAR_JOG Moves TCF in a straight line in a specified direction until an asynchronous event occurs (e.g. an asynchronous error or a call to CxStopMechanism).

A path move or a circular move overrides this setting.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
CxGetInvkinSoln

Gets the current inverse kinematics solution number setting

SYNOPSIS

#include <code/robpac.h>
long CxGetInvkinSoln (CxMechanism mech, long *soln_num)

ARGUMENTS

mech    Mechanism ID;
This is a CxMechanism ID returned by a call to CxOpenMechanism.
soln_num Inverse kinematic solution number

DESCRIPTION

This function gets the mechanism’s current inverse kinematic solution setting. If the solution number is set to CX_ALL_SOLN (-1) all possible solutions will be calculated during a move; otherwise, the mechanism will use the inverse kinematic solution specified by the solution number.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxSetInvkinSoln, CxGetActualInvkinSoln
CxGetJacobian

Gets the current setting for the Jacobian speed checking flag

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetJacobian(CxMechanism mech, long *flag)
```

ARGUMENTS

- `mech` Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `flag` The current value for the Jacobian speed checking flag (CX_ON or CX_OFF).

DESCRIPTION

This function gets the value for the current Jacobian checking flag. Mathematically, the inverse Jacobian maps Cartesian velocities onto the joint rates of the mechanism. When the Jacobian flag is set to CX_ON, it specifies that the joint speeds will be checked. The rates of those joints, which hit their limits will be reduced by CIMServer default procedures or by some user-provided function.

It is useful to disable Jacobian flag checking for mechanisms, which have redundant joints. It is also recommended that when the user does not have an accurate model of the way the controller handles the excessive joint rates during path-following moves. The default setting is CX_ON.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

HARDWARE AND SYSTEM DEPENDENCIES

It is often difficult to determine how controller vendors treat the joint control of joints that are colinear (redundant). In the absence of an accurate model, it may be useful to turn Jacobian checking off and assume that the cycle times are only approximate. Otherwise, the CIMServer’s Jacobian and speed utilities may interrupt or halt the motion sequences.

SEE ALSO

CxSetJacobian
CxGetJointMoveTime

Determines the time to make a joint move

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetJointMoveTime(CxMechanism mech, CxTimedMove *move,
                        double *dof1, double *dof2, *jnt_slow, double *move_time)
```

ARGUMENTS

- **mech**: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **move**: Move data structure that has been loaded with the motion parameters.
- **dof1**: Joint values for the initial configuration of the mechanism.
- **dof2**: Joint values for the final configuration of the mechanism.
- **jnt_slow**: Joint number of the slowest joint in the joint interpolated move from initial joint values to final joint values.
- **move_time**: Time desired for the next joint move.

DESCRIPTION

This function gets the move time for the next move (and only the next move) which follows this function. The next move must be a joint move. The move data structure must be loaded with valid motion parameters for the mechanism and can be loaded with the parameters currently stored in the server database by using the function CxLoadMoveParameters. As an alternative, the user can set these parameters directly by referring to the data structure definition, which follows, being careful to use the correct units as set by CxSetUnit (normally using degrees for rotational joints and mm for translational joints). The function CxSetJointMoveTime can be used to set the desired move time for the next joint move, which follows. The move data structure is defined as:

```c
typedef struct CxTimedMove
{
    long traj_accel_type;  /* get from CxGetTrajAccelType */
    long accel_type;       /* get from CxGetAccelType */
    double jnt_set;        /* get from CxGetJointSpeed */
    /* get from CxGetMaxJntSpeed */
    double jnt_speed_max[CX_MAXJOINTS];

    /* get from CxGetJntAccelMax */
    double jnt_accel_max[CX_MAXJOINTS];

    /* get from CxGetJntAccelMax */
    double jnt_decel_max[CX_MAXJOINTS];

    double jnt_accel[CX_MAXJOINTS];    /* get from CxGetJntAccel */
    double jnt_decel[CX_MAXJOINTS];    /* get from CxGetJntAccel */
};
```
double rise_S_time; /* get from CxGetSAccelTimes */
double fall_S_time; /* get from CxGetSAccelTimes */
double rise_time_min; /* get from CxGetAccelTimesMin */
double fall_time_min; /* get from CxGetAccelTimesMin */
double rise_time; /* get from CxGetTrapAccelTimes */
double fall_time; /* get from CxGetTrapAccelTimes */
} CxTimedMove;

The user can use this function to determine the required times to make certain joint moves, but should be aware that the trajectory calculations in the server are governed by the current motion parameters. To achieve the desired time, the user must ensure that the motion parameters used in the move time calculations must be the same as those in the server data base. The user can change these settings by using the “Set” counterparts of the “Get” functions mentioned in the comments of the data structure definition.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following code will get the move time for a joint move from joint values specified in dof1 to joint values specified in dof2. This move time will then be specified for this move. Note that the move data structure argument must be entered as an address.

CxMechanism mech;
CxTimedMove move;
CxNodeId rob_id;
long jnt_slow;
double move_time;
double dof1[CX_MAXJOINTS], dof2[CX_MAXJOINTS];

dof1[0] = dof1[1] = dof1[2] = 0.0;
dof2[0] = 125.3;
dof2[1] = 23.2;
dof2[2] = -300.2;
CxMoveAllAxes(mech, dof1);
CxLoadMoveParameters(mech, rob_id, &move);
CxGetJointMoveTime(mech, &move, dof1, dof2, &jnt_slow, &move_time);
WARNINGS

This function does not take into account the MEI-DSP specific “short move policies” for s-curve moves. Hence, the move time computed for s-curves on the MEI-DSP motion card will only be approximate. Details on the short move policies can be found in the MEI-DSP Device Interface release notes and the Cimetrix engineering report, *Analysis of Short Move Acceleration*.

SEE ALSO

CxSetJointMoveTime
**CxGetJointSpeed**

Gets the current mechanism's joint interpolated speed setting

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetJointSpeed(CxMechanism mech, double *speed)
```

**ARGUMENTS**

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **speed**: Current percent joint speed. The value returned is between 0.0 and 1.0, where 0.5 would represent 50% of the mechanism's maximum speed.

**DESCRIPTION**

This function gets the percentage of a mechanism's maximum joint speed that will be used in subsequent joint moves. The joint speed setting is used only when the motion type is set to CX_JOINT_INTERP, otherwise, the tip and screw speeds are used. The joint speed has no effect on non-joint-interpolated moves.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

CxSetJointSpeed, CxGetActualJointSpeed, CxSetTipSpeed, CxGetTipSpeed, CxSetScrewSpeed, CxGetScrewSpeed
CxGetLatchedAxes

Gets the latched joint values for the given mechanism

SYNOPSIS
#include <code/robpac.h>
#include <oac/oac.h>
long CxGetLatchedAxes(CxMechanism mech, double *value)

ARGUMENTS
mech  Mechanism ID;
      This is a CxMechanism ID returned by a call to CxOpenMechanism.
value  An array of doubles to store the latched joint values.

DESCRIPTION
This function gets the latched joint values for the given mechanism. It should be called after the
CxLatchOnTrigger function has been called and the trigger condition has occurred. In order to use this
function, mech should be opened with CX_CONTROL mode. If the CIMServer is in simulation mode,
CxGetLatchedAxes will return the current joint values in the simulation model. The latch buffer is
cleared after the CxGetLatchedAxes is called, i.e., CxGetLatchedAxes can only be called once after
the latch condition is met and the joint values are latched. The user has to allocate memory space for the
value array. The size of this array should be at least the same size as the number of joints of the given
mechanism.

At present, this function is only supported on the PMAC and MEI DSP motion cards.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MACHINE_OUT_OF_MEMORY</td>
<td>No memory is available for the value array.</td>
</tr>
<tr>
<td>OAC_NO_MOTOR_LATCHED or MEI_POS_NOT_LATCHED</td>
<td>No position has been latched yet.</td>
</tr>
</tbody>
</table>

EXAMPLE
This example demonstrates how to get the latched joint positions from CIMControl.

...
CxServer my_server;
Long latch_signal, value=0;
CxMechanism my_mech;
Double latch_value[MAXJOINTS];
.

/* my_mech has been opened for CX_CONTROL prior to this code segment */
latch_signal = SGI_10;
CxLatchOnTrigger(my_server, latch_signal, my_mech, CXLATCH_RISE_FLAG);
.
.
CxWaitForValue(my_server, SIG_10, 1, 0);
CxGetLatchedAxes(my_mech, latch_value);
.
.
SEE ALSO
CxSetJointSpeed, CxGetActualJointSpeed, CxSetTipSpeed, CxGetTipSpeed,
CxSetScrewSpeed, CxGetScrewSpeed
CxGetMotionStatus

Gets the mechanism’s current motion status

SYNOPSIS

#include <code/robpac.h>
long CxGetMotionStatus(CxMechanism mech, long *mstat)

ARGUMENTS

mech Mechanism ID;
   This is a CxMechanism ID returned by a call to CxOpenMechanism.

mstat Current mechanism motion status. See DESCRIPTION below for possible values.

DESCRIPTION

This function gets a mechanism’s current motion status. The possible motion status parameters are defined as follows:

CX_START_MOTION At start of a new motion segment.
CX_IN_MOTION In motion on a single segment.
CX_END_OF_MOTION At end of motion on a single segment.
CX_NOT_IN_MOTION Motion segment not yet started.
CX_NO_MOTION Single segment has no move distance.
CX_IN_MOTION_IN In motion during a blend of two segments.
CX_IN_MOTION_END Still in motion on first segment but finish move on second segment during a blend.
CX_END_MOTION_IN End motion on first segment but still in motion on second segment during a blend.
CX_END_MOTION_END Both segments are finished during a blend.
CX_IN_MOTION_NO Moving on first segment and the second segment has no move distance during a blend.

CX_RESUME_MOTION A halted motion is being restarted.
CX_NO_MOTION_PENDING The current move is finished, and the next element in the queue is not a motion related function.
CX_INTERRUPT_TELE The current motion has been interrupted and the mechanism placed in tele-operated mode.
CX_ABORT_MOTION_END The motion has been aborted, and motion on the current segment has reached zero velocity.
CX_ABORT_MOTION_ABORT Motion is aborted during a blend and velocity is zero.
CX_ABORT_MOTION_IN Motion during a blend has reached zero velocity for the first segment but not the second.
CX_IN_MOTION_ABORT Motion during a blend has reached zero velocity for the second segment but not the first.

Any of these constants can be in effect when the CIMServer is generating the trajectories. However, when the trajectories are computed on the motion card rather than within CIMControl (as with most non-blended, joint-interpolated moves), only CX_IN_MOTION and CX_NOT_IN_MOTION can be in effect.
RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
**CxGetMotionTrack**

Gets the current value of the motion tracking flag

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetMotionTrack(CxMechanism mech, long *flag)
```

**ARGUMENTS**

- `mech` Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `flag` Value of the motion tracking flag

**DESCRIPTION**

This function gets the value of the motion tracking flag. If motion tracking is `CX_ON`, the mechanism will continually update the target value from the database and converge to the moving target (if the mechanism speed is higher than the target speed). In effect, the CIMServer’s initialization routines are called for each trajectory segment and will slow down the trajectory calculations. The mechanism will not track targets physically attached to the mechanism even if tracking is turned on. The default setting is `CX_OFF`.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program tries to chase a moving object. First, it turns motion tracking on. Then it moves the TCF to the moving object. As soon as the TCF gets there, the program picks up the moving target and sets tracking to `CX_OFF`. Then it gets the tracking setting and verifies that it is set to `CX_OFF`.

```c
#include <stdio.h>
#include <code/robpac.h>

void pickup_part(); /* This function is used to pick up a target */

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100, moving_target, tcf;
    long      is_still_tracking;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
```
mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
CxSetBlendPolicy ( mech, CX_MOVE_WAIT );

/* start the target in motion */
/* Chase moving object and pick it up*/
CxSetMotionTrack ( mech, CX_ON );
CxMoveRelNode ( mech, moving_target, tcf, "XYZ", 0.0, 0.0, 0.0,
5.0, 5.0, 5.0 );
pickup_part(); /* function used to pick up the target */
CxSetMotionTrack(mech, CX_OFF);
CxGetMotionTrack ( mech, &is_still_tracking );
if  ( is_still_tracking == CX_ON )
{
   CxStopMechanism ( mech );
   printf(" Problem with tracking. Aborting move... \n");
}

CxRobpacExit();

WARNINGS
Motion tracking slows down the trajectory calculations. It should only be used when you wish to track a
particular moving target; thus, it is wise to enable motion tracking only before moving toward a target. After
move completion, you should turn the tracking off.

SEE ALSO
CxSetMotionTrack
**CxGetMoveBufferSize**

Gets the buffer size of the move buffer on the servocard

**SYNOPSIS**

```
#include <code/robpac.h>
long CxGetMoveBufferSize(CxMechanism mech, long *size)
```

**ARGUMENTS**

- **mech** Mechanism ID;
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **size** Servocard buffer size currently being used.

**DESCRIPTION**

This function gets the current buffer size being used on the controller card. Buffers are used to queue a sequence of moves so that the card can blend these moves together or improve cycle times by reducing the time latency between adjacent moves. Usually, this buffer size will be set to some number between 2 and 10.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

CxSetMoveBufferSize
**CxGetMoveTangent**

Gets the value for which two moves are considered tangent

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetMoveTangent(CxMechanism mech, double *tang_value)
```

**ARGUMENTS**

- `mech` Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `tang_value` Tangent value in either degrees or radians, depending on current units setting

**DESCRIPTION**

This function gets the value at which two moves are considered tangent. This function is used along with the special blend policies of `CX_MOVE_TO_TANGENT` or `CX_MOVE_THRU_TANGENT`, which seek to maintain the exact path and speed setting for paths that are tangent to each other. In contrast to the blend policy of `CX_MOVE_THRU`, motion between tangent moves will not blend the decel period of the first move with the accel period of the second move. Normally, the user would not want to set the tangent value to more than 5 degrees; otherwise significant path deviation will occur. Currently, this function can only be used in evaluating tangency between `curveseg` moves and NC feed moves.

A move in which the tool orientation dominates curvilinear displacement (position change) will be considered non-tangent to the previous move.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program gets the tangent value. If it is greater than one degree, its value is reset to one degree.

```c
#include <stdio.h>
#include <code/robpac.h>
void pickup_part();

void main(void)
{
    double tang_value;
    CxServer     Server;
```
CxMechanism mech;
CxNodeId   s100, path, tcf;

Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
.
mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;

CxGetMoveTangent ( mech, &tang_value ) ;

if ( tang_value > 1.0 )
   CxSetMoveTangent ( mech, 1.0 ) ;
.

CxRobpacExit ( ) ;
}

SEE ALSO

CxSetMoveTangent
**CxGetNumConfig**

Gets the number of configurations taught for a given mechanism

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetNumConfig(CxMechanism mech, long *num)
```

**ARGUMENTS**

- `mech` Mechanism ID;
  - This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `num` Number of configurations taught for the given mechanism

**DESCRIPTION**

This function gets the number of configurations taught for a given mechanism.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following process gets the total number of configurations taught for mechanism s100, then lists the configuration names to stderr.

```c
#include <stdio.h>
#include <code/robpac.h>
#include <code/cntr_const.h>

void main ( void )
{
    CxServer peg;
    CxMechanism s100;
    long i, num;
    char name[CX_MAXNAME];
    CxNodeId s100_id;

    /* open server */
    peg = CxOpenServer("Peg", CX_SMEM, 0);
    CxGetNamedNodeId(peg, "s100", &s100_id);
```
/* open mechanism */
CxSetControllerType(peg, s100_id, CX_PMAC_DRVR);
s100 = CxOpenMechanism(peg, s100_id, CX_CONTROL);

CxSetErrorPolicy(CX_RETURN_ERRORS);

CxGetNumConfig(s100, &num);

for(i=0; i<num; i++)
{
    CxGetConfigName(s100, i, name);
    fprintf(stderr,"configuration %ld: %s\n", i+1, name);
}

CxRobpacExit();

SEE ALSO
   CxGetConfigName, CxListConfig, CxMoveToConfig, CxTeachConfig, CxDeleteConfig
CxGetPercentBlendSpeed

Gets the percent of set speed at which blending begins

SYNOPSIS

#include <code/robpac.h>
long CxGetPercentBlendSpeed (CxMechanism mech, double *fraction)

ARGUMENTS

mech Mechanism ID;
This is a CxMechanism ID returned by a call to CxOpenMechanism.

fraction The fractional percent of full speed at which blending of two moves begins

DESCRIPTION

This function will get the fractional percent of full speed at which blending of two moves will occur. For example, if the fractional percent setting is 0.8, then a second move will begin to blend with the first move when the latter has decelerated to 80% of its maximum move speed.

See the CODE Applications Programming manual for a description of blended and non-blended moves.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX.MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX.MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX.INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program sets the tool speed to 100 mm/s, then sets the blend type to CX_PERCENT_SPEED_BLEND, and then uses the CxGetPercentBlendSpeed() to determine the current setting for the percent blend speed. If it is less than 0.5 (i.e. 50%), then its value is set to 0.5. This means that the blending of two moves will begin when the speed of the first move falls below 50 mm/s.

#include <stdio.h>
#include <code/robpac.h>

void main ( void )
{
    CxServer Server ;
    CxMechanism mech ;
    CxNodeId s100 ;
    double fraction ;
Server = CxOpenServer( "Testing", CX_SMEM, 0 ) ;

mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;
CxSetToolMotionType ( mech, CX_LINEAR_INTERP) ;
CxSetTipSpeed ( mech, 100.0 ) ;
CxSetBlendPolicy ( mech, CX_MOVE_THRU ) ;
CxSetBlendType ( mech,CX_PERCENT_SPEED_BLEND ) ;
CxGetPercentBlendSpeed ( mech, &fraction ) ;

if ( fraction < 0.5 )
   CxSetPercentBlendSpeed ( mech, 0.5 ) ;

CxWaitForEndOfMotion ( mech ) ;

CxRobpacExit ( ) ;

WARNINGS
The user must be sure that the motion card is capable of performing this type of blended move. Currently, CIMControl only supports blending on the PMAC and MEI-DSP motion cards.

HARDWARE AND SYSTEM DEPENDENCIES
Move types and blending capabilities depend on the controller hardware and software. It is not safe to assume that all controllers support these move capabilities. In CODE 4.0, CIMControl supports blended moves on the PMAC and MEI-DSP cards, but not on the MEI XMP motion cards.

SEE ALSO
CxGetBlendPolicy,CxSetPercentBlendSpeed,CxSetBlendType
CxGetSAccelTimes

Gets the S ramp acceleration and deceleration times.

SYNOPSIS

#include <code/robpac.h>
long CxGetSAccelTimes(CxMechanism mech, double *rise_S_time, double *fall_S_time)

ARGUMENTS

mech Mechanism ID;
This is a CxMechanism ID returned by a call to CxOpenMechanism.
rise_S_time Amount of time for constant jerk portion of the acceleration phase.
fall_S_time Amount of time for constant jerk portion of the deceleration phase.

DESCRIPTION

This function is used to determine the current S-curve acceleration and deceleration times. These times are used if the mechanism’s current acceleration type is CX_CONST_RAMP_TIME (see CxGetAccelType, CxSetAccelType). An S-curve acceleration profile is useful for minimizing the wear on a mechanism because the forces being applied to the mechanism’s gear train are changed gradually through the acceleration period.

When a CX_CONST_RAMP_TIME acceleration setting is used, the S-curve acceleration and deceleration profiles are combined with the constant trapezoidal acceleration and deceleration times respectively (see CxGetTrapAccelTimes, CxSetTrapAccelTimes).

The following illustrations show the relationship of the trapezoidal acceleration times and the S curve acceleration times. The figures include motion with no S-curve acceleration (pure trapezoidal acceleration profile), pure S-curve acceleration, and combined S-curve and trapezoidal acceleration profiles. Note that in these figures, the acceleration profiles are the same as the deceleration profiles (i.e. the acceleration times are the same as the deceleration times); therefore, S_time is used to represent both. Likewise, rise_time and fall_time are used to represent trapezoidal acceleration times.

In order for the S-curve acceleration parameter to be used when operating in CX_RUNTIME mode, the underlying motion control interface must support S-curve acceleration profiles (see HARDWARE AND SYSTEM DEPENDENCIES below).
NOTE: If the specified value for rise_S_time is greater than \( \frac{1}{2} \) the rise_time, the rise_S_time will be set to \( \frac{1}{2} \) the rise_time.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

HARDWARE AND SYSTEM DEPENDENCIES

On the MEI-DSP motion card, the acceleration profile is always the same as the deceleration profile. In this case only the rise_S_time parameter is used, and the fall_S_time parameter is ignored.

SEE ALSO

CxSetSAccelTimes, CxGetTrapAccelTimes
CxGetScrewSpeed

Gets the current screw speed setting used in curvilinear moves

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetScrewSpeed(CxMechanism mech, double *speed)
```

ARGUMENTS

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **speed**: Current screw speed setting in deg/sec or rad/sec, depending on the units specified via CxSetUnit.

DESCRIPTION

This function is used to get the current screw speed setting (used in curvilinear moves) for a given mechanism. The higher the screw speed, the faster a tool will rotate.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program sets up the motion parameters and moves the TCF of the mechanism to various positions. Next, it gets the trajectory rate and prints it out. Finally, it continuously gets the values of the velocity vector, the tool speed, the screw speed and prints them out.

```c
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf, t1, t2, t3;
    double    screw_speed, tip_speed
    double    traj_rate;
    CxVector    angle, position;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );
```
mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO);
CxSetInterpType( mech, CX_LINEAR_INTERP );

CxMoveToNode( mech, t1, tcf );
CxMoveToNode( mech, t2, tcf );
CxMoveToNode( mech, t3, tcf );

CxGetTrajectoryRate( mech, &traj_rate );
printf(" Current trajectory rate : %lf \n", traj_rate );
while (1)
{
    CxGetToolSpeedVec( mech, MyMech, angle, position );
    printf(" Current angular velocity components : [ %lf %lf %lf ] \n",
            angle[0], angle[1], angle[2]);
    printf(" Current angular velocity components : [ %lf %lf %lf ] \n",
            position[0], position[1], position[2]);

    CxGetTipSpeed( mech, &tip_speed );
    printf(" Current tip speed : %lf \n", tip_speed );

    CxGetScrewSpeed( mech, &screw_speed );
    printf(" Current screw speed : %lf \n", screw_speed );
    ...
}

SEE ALSO
CxGetActualScrewSpeed CxGetScrewSpeed CxSetScrewSpeed, CxSetUnit
CxGetTipSpeed

Gets the current linear speed setting in curvilinear moves

SYNOPSIS

#include <code/robpac.h>
long CxGetTipSpeed(CxMechanism mech, double *speed)

ARGUMENTS

mech  Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
speed  Current vector speed of the tool

DESCRIPTION

This function gets the current linear speed setting for a mechanism’s tool control frame. This value is ignored for moves with the interpolation type set to CX_JOINT_INTERP.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program sets up the motion parameters and moves the TCF of the mechanism to various positions. Next, it gets the trajectory rate and prints it out. Then it continuously gets the values of the velocity vector, the tip speed, and the screw speed and prints them out.

#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf, t1, t2, t3;
    double    screw_speed, tip_speed
    double    traj_rate;
    CxVector    angle, position;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );
    //
    //
mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO);
CxSetInterpType( mech, CX_LINEAR_INTERP );

CxMoveToNode( mech, t1, tcf );
CxMoveToNode( mech, t2, tcf );
CxMoveToNode( mech, t3, tcf );

CxGetTrajectoryRate( mech, &traj_rate );
printf(" Current trajectory rate : %lf \n", traj_rate );
while (1)
{
    CxGetToolSpeedVec( mech, MyMech, angle, position );
    printf(" Current angular velocity components : [ %lf %lf %lf ] \n",
            angle[0], angle[1], angle[2]);
    printf(" Current angular velocity components : [ %lf %lf %lf ] \n",
            position[0], position[1], position[2]);

    CxGetTipSpeed( mech, &tip_speed );
    printf(" Current tip speed : %lf \n", tip_speed );

    CxGetScrewSpeed ( mech, &screw_speed );
    printf(" Current screw speed : %lf \n", screw_speed );
    .
    .
}

SEE ALSO
    CxSetInterpType, CxSetTipSpeed
**CxGetToolMotionType**

Gets the tool motion type setting

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetToolMotionType(CxMechanism mech, long *tool_motion_type)
```

**ARGUMENTS**

- `mech` Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `tool_motion_type` Tool motion type setting. See DESCRIPTION below for possible tool motion types.

**DESCRIPTION**

This function is used to get the tool motion type setting. This parameter defines how a TCF will align itself with a target to which it moves. For a more detailed description of the tool motion type, refer to the CODE Applications Programming manual. Currently, a mechanism can have one of the four tool motion types:

- **CX_FULL_POSE**: Tool must align exactly as specified (the default).
- **CX_FIXED_ORIENT**: Tool must maintain its current orientation.
- **CX_Z_POSE**: Tool Z interpolation occurs in plane parallel to tool and target Z axes.
- **CX_Z_POSE_NO_SPIN**: Tool Z interpolation occurs in plane parallel to tool and target Z axes but no interpolation spin about Z axis to align tool X,Y axes with target X,Y axes.

A **CX_SERIAL_ROBOT** is treated differently from a **CX_NC_ROBOT** when using the **Z_POSE** type. The **CX_SERIAL_ROBOT** assumes that the tool can be spun around the tool Z axis to align the tool X,Y axes with the target X,Y axes. The **CX_NC_ROBOT** cannot; thus, if the mechanism is **CX_NC_ROBOT** and **CX_Z_POSE** is specified, then the type defaults to **CX_Z_POSE_NO_SPIN**.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a mechanism for **CX_CONTROL**, then checks the interpolation type. If the type is set to **CX_LINEAR_INTERP**, it then checks for the tool motion type. If the tool motion type is set to **CX_FULL_POSE**, it is reset to **CX_FIXED_ORIENT**.
```c
#include <code/robpac.h>

void main(void)
{
    CxServer       Server;
    CxMechanism    mech;
    long           interp_type, tool_motion_type;
    CxNodeId       MyMech, tcf, t1;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId ( Server, "MyMech", &MyMech );
    CxGetNamedNodeId ( Server, "tcf", &tcf );
    CxGetNamedNodeId ( Server, "t1", &t1 );
    
    mech = CxOpenMechanism ( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy ( mech, CX_MOVE_TO );
    
    CxGetInterpType ( mech, &interp_type );
    if (interp_type == CX_LINEAR_INTERP)
    {
        CxGetToolMotionType ( mech, &tool_motion_type );
        if (tool_motion_type == CX_FULL_POSE)
            CxSetToolMotionType ( mech, CX_FIXED_ORIENT );
    }
    CxMoveToNode ( mech, t1, tcf );
    
    CxWaitForEndOfMotion ( mech );
    CxRobpacExit ();
}

SEE ALSO
    CxSetToolMotionType
```
CxGetToolName

Gets the current tool TCF being used by the mechanism

SYNOPSIS

#include <code/robpac.h>
long CxGetToolName (CxMechanism mech, CxNodeId node, char *name)

ARGUMENTS

mech     Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
node     Mechanism node ID
name     TCF name

DESCRIPTION

This function gets the current name of the TCF being used by the mechanism. This is the last joint frame if
the most recent move was not an inverse kinematics move; otherwise, it is the TCF used in the move. The
node must have the ROBOT attribute.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given node does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
<tr>
<td>CX_ROB_HAS_NO_JOINTS</td>
<td>There is no tool for the given mechanism.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_ATTRIBUTE_NOT_FOUND</td>
<td>Node does not have the ROBOT attribute.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program gets the name of the tool used by the mechanism and prints it out.
#include <code/robpac.h>

void main (void)
{
    CxNodeId MyMech, peg_1_loc, grip_tcf
    CxServer    Server;
    CxMechanism mech;
    char      tool_name[CX_MAXNAME];

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId ( Server, "s100", &MyMech);
    CxGetNamedNodeId ( Server, "peg_1_loc", &peg_1_loc);
}
CxGetNamedNodeId (Server, "grip_tcf", &grip_tcf);
mech = CxOpenMechanism (Server, MyMech, CX_CONTROL);

CxGetToolName (mech, MyMech, tool_name);
printf("MyMech most recently used tool named %s \n", tool_name);
/* no move was made so this will be the last joint frame */
.
.
CxMoveToNode (mech, peg_1_loc, grip_tcf);
CxGetToolName (mech, MyMech, tool_name);
printf("Tool most recently used is called %s \n", tool_name);
/* move was not invkin so this will be last joint frame */
.
.
}
CxGetToolSpeedVec

Gets the current pose velocity components of the TCF frame

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetToolSpeedVec (CxMechanism mech, CxNodeId node, CxVector angles, CxVector vec)
```

**NOTE:** Type CxVector is defined in `<code/matx_def.h>` as follows:
```c
typedef double CxVector[3];
```

ARGUMENTS

- **mech** Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **node** Mechanism node ID. This is an ID returned by a call to CxOpenMechanism.
- **angles** Contains three (x, y, z) angular velocity components.
- **vec** Contains three (x, y, z) linear velocity components.

DESCRIPTION

This function retrieves angular and linear velocity components representing the TCF frame pose velocity as resolved into the mechanism base frame. The results will only have meaning if a mechanism is in motion and performing an inverse kinematics move.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given node does not exist.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_ATTRIBUTE_NOT_FOUND</td>
<td>Node does not have the ROBOT attribute.</td>
</tr>
</tbody>
</table>

EXAMPLE

This program sets up the motion parameters and moves the TCF of the mechanism to various positions, then it gets the trajectory rate, prints it out, and continuously gets the values of the velocity vector, the tool speed, the screw speed and prints them out.
```c
#include <code/robpac.h>

void main (void)
```

{
    CxServer Server;
    CxMechanism mech;
    CxNodeId MyMech, tcf, t1, t2, t3;
    double screw_speed, tip_speed
    double traj_rate;
    CxVector angle, position;
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );

    mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO);
    CxSetInterpType( mech, CX_LINEAR_INTERP );

    CxMoveToNode( mech, t1, tcf );
    CxMoveToNode( mech, t2, tcf );
    CxMoveToNode( mech, t3, tcf );

    CxGetTrajectoryRate( mech, &traj_rate );
    printf(" Current trajectory rate : %lf \n", traj_rate );
    while (1)
    {
        CxGetToolSpeedVec( mech, MyMech, angle, position );
        printf(" Current angular velocity components : [ %lf %lf %lf ]
", angle[0], angle[1], angle[2]);
        printf(" Current angular velocity components : [ %lf %lf %lf ]
", position[0], position[1], position[2]);

        CxGetTipSpeed( mech, &tip_speed );
        printf(" Current tip speed : %lf \n", tip_speed );

        CxGetScrewSpeed ( mech, &screw_speed );
        printf(" Current screw speed : %lf \n", screw_speed );
    }
}
}
CxGetTrajAccelType

Sets the type of trajectory acceleration or deceleration profile

SYNOPSIS
#include <code/robpac.h>
long CxGetTrajAccelType(CxMechanism mech, long *type)

ARGUMENTS
mech  Mechanism ID
This is a CxMechanism ID returned by a call to CxOpenMechanism.
type  Current trajectory profile type of mech. See DESCRIPTION below for definitions for possible types.
vec   Returns the relative position vector.

DESCRIPTION
This function retrieves the current acceleration and deceleration profile type of a mechanism. The profile describes the shape of the velocity versus time curve as the mechanism changes either Cartesian or joint speed. The default type is CX_TRAP_ACCEL in which the speed changes linearly with time. The CX_S_ACCEL type uses constant jerk S sections to transition the speed change smoothly. Currently, two profile types can be set for mechanism motion.

CX_TRAP_ACCEL: Trapezoidal profile.
CX_S_ACCEL: Uses constant jerk S transitions to change speed.

If the accel type is specified as CX_CONST_RAMP_ACCEL and the profile type is CX_S_ACCEL, then the time to perform the speed change is calculated using the ramp acceleration set for this move. If the accel type is CX_CONST_RAMP_TIME and the profile type is CX_S_ACCEL, then the S accel times set by the API function CxSetSAccelTimes() are compared to the ramp time set by the CxSetTrapAccelTimes(). The S transition times will be compared to the ramp time and if they exceed the ramp time, the ramp time will be increased to the S times. In this case, no linear transition will exist.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is not CX_TRAP_ACCEL or CX_S_ACCEL.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
The following program opens a mechanism for CX_CONTROL, then gets the current trajectory accel type.

```c
#include <code/robpac.h>

void main(void)
{
    CxServer   Server;
    CxMechanism mech;
    long type;
    CxNodeId   MyMech;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeid( Server, "MyMech", &MyMech );
    .
    mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );

    CxGetTrajAccelType( mech, &type);
    .
    CxRobpacExit();
}

SEE ALSO
CxSetTrajAccelType
CxGetTrajectoryRate

Gets the trajectory rate setting for Cartesian path following

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetTrajectoryRate(CxMechanism mech, double *rate)
```

ARGUMENTS

- mech: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- rate: Rate in seconds.

DESCRIPTION

This function gets the trajectory rate setting at which the mechanism’s kinematics are calculated along paths in Cartesian space. Mechanisms typically use values of 0.1 seconds (10 Hz) to 0.01 seconds (100 Hz), depending upon the mechanism kinematic complexity — the lower the setting (i.e. the higher the frequency), the more accurate the path following.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

WARNINGS

The user must be extremely cautious when modifying the trajectory rate if it will be used to actually control a mechanism. If the speed of the CIMServer’s kinematic calculations (forward and inverse) cannot be completed in this time interval, then the data sent to the physical controller will lag the rates expected by the hardware. This delay could cause unpredictable motion responses from the mechanism and other mechanisms being controlled resulting in damage to the mechanisms or surrounding equipment. The cautious user should first enter settings that are high and then decrease them slowly for all applications that are being programmed.

SEE ALSO

- CxSetTrajectoryRate
**CxGetTrapAccelRamps**

Gets the rise and fall acceleration settings for trapezoidal motion profiles

**SYNOPSIS**

#include <code/robpac.h>

long CxGetTrapAccelRamps(CxMechanism mech, double *rise_ramp, double *fall_ramp)

**ARGUMENTS**

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **Rise_ramp**: Positive acceleration value in linear units (mm/s^2, in/s^2, etc.)
- **Fall_ramp**: Positive deceleration value in linear units (mm/s^2, in/s^2, etc.)

**DESCRIPTION**

The function allows the user to get the rise and fall acceleration rate settings for trajectory following in Cartesian space where trapezoidal motion profiles are used. Acceleration rates are used in motion calculations when the acceleration type is set to CX_CONST_RAMP_ACCEL (see CxGetAccelType, CxSetAccelType).

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first sets the motion related parameters. Next, it sets the velocity profile type to CXCONST_RAMP_ACCEL. Finally, it gets the current settings for the acceleration and deceleration values and resets them if appropriate.

```c
#include <code/robpac.h>

#define MAXACCEL  0.1
#define MAXDECEL  0.1

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf, t1, t2, t3;
    double    rise_time, fall_time;
```
double    accel, decel;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxGetNamedNodeId ( Server, "MyMech", &MyMech );
CxGetNamedNodeId ( Server, "tcf", &tcf );
CxGetNamedNodeId ( Server, "t1", &t1 );
.
.
mech = CxOpenMechanism ( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy ( mech, CX_MOVE_TO);
CxSetInterpType ( mech, CX_LINEAR_INTERP );

/* Set motion velocity and acceleration profile parameters */
CxSetAccelType ( mech, CX_CONST_RAMP_ACCEL );

/* Check velocity and acceleration profile parameters */
CxGetTrapAccelRamps ( mech, &accel, &decel );
if ( ( accel>MAXACCEL) || ( decel> MAXDECEL ) )
  CxSetTrapAccelRamps ( mech, MAXACCEL*0.7, MAXDECEL*0.7 );
CxMoveToNode ( mech, t1, tcf );
.
.
CxWaitForEndOfMotion ( mech );
CXRobpacExit ();
}

**CxGetTrapAccelTimes**

Gets the rise and fall time settings for trapezoidal motion profiles

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetTrapAccelTimes(CxMechanism mech, double *rise_time, double *fall_time)
```

**ARGUMENTS**

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **rise_time**: Rise acceleration time setting in seconds.
- **fall_time**: Fall deceleration time setting in seconds.

**DESCRIPTION**

This function is used to get the current trapezoidal acceleration and deceleration times. These times are used if the mechanism’s current acceleration type is `CX_CONST_RAMP_TIME` (see CxGetAccelType, CxSetAccelType). These values may be set lower than the minimum accel times, but when they are used to calculate trajectory, the minimum accel times will be used (see CxSetAccelTimesMin).

When a `CX_CONST_RAMP_TIME` acceleration setting is used, S-curve acceleration and deceleration profiles can be combined with the constant trapezoidal acceleration and deceleration times respectively (see CxGetSAccelTimes, CxSetSAccelTimes). An S-curve acceleration profile is useful for minimizing the wear on a mechanism because the forces being applied to the mechanism’s gear train are changed gradually through the acceleration period.

The following illustrations show the relationship of the trapezoidal acceleration times and the S curve acceleration times. The figures include motion with no S-curve acceleration (pure trapezoidal acceleration profile), only S-curve acceleration, and combined S-curve and trapezoidal acceleration profiles. Note that in these figures, the acceleration profiles are the same as the deceleration profiles (i.e. the acceleration times are the same as the deceleration times); therefore, `S_time` is used to represent both. Likewise, `rise_time` and `fall_time` are used to represent trapezoidal acceleration times.

In order for the S-curve acceleration parameter to be used under CIMControl, the underlying motion control interface must support S-curve acceleration profiles (see HARDWARE AND SYSTEM DEPENDENCIES below).
NOTE: If the specified value for rise_S_time is greater than \( \frac{1}{2} \) the rise_time, the rise_S_time will be set to \( \frac{1}{2} \) the rise_time.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program first sets the motion related parameters and sets the ramp acceleration type to be of CX_CONST_RAMP_ACCEL. Then it gets the current settings for the rise and fall times and resets them if appropriate.

```c
#include <code/robpac.h>
#define MIN_RISE_TIME 1.0
#define MIN_FALL_TIME 1.0

void main (void)
{
    CxServer   Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf, t1, t2, t3;
    
    // Set motion related parameters
    
    // Get current settings for rise and fall times
    
    // Reset the settings if appropriate
    
    // Additional code for handling errors
}
```
double rise_time, fall_time;
double accel, decel;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxGetNamedNode( Server, "MyMech", &MyMech );
CxGetNamedNode( Server, "tcf", &tcf );
CxGetNamedNode( Server, "t1", &t1 );

mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO);
CxSetInterpType( mech, CX_LINEAR_INTERP );

/* Set motion velocity and acceleration profile parameters */
CxSetAccelType( mech, CX_CONST_RAMP_ACCEL );

/* Check velocity and acceleration profile parameters */
CxGetTrapAccelTimes( mech, &rise_time, &fall_time );
if ( (rise_time<MIN_RISE_TIME) || (fall_time<MIN_FALL_TIME) )
    CxSetTrapAccelTimes(mech, MIN_RISE_TIME*1.5,
                      MIN_FALL_TIME*1.5 );

CxMoveToNode( mech, t1, tcf );

CxWaitForEndOfMotion( mech );
CxRobpacExit();

WARNINGS

The user must be extremely cautious when using these values if they will be used to actually control a
mechanism since allowable accelerations are mechanism and design-dependent. Excessive accelerations may
cause the mechanism’s inertial loads to damage the drive transmissions and other mechanical components on
the mechanism.

HARDWARE AND SYSTEM DEPENDENCIES

When using an MEI-DSP motion card, the acceleration profile is always the same as the deceleration profile,
so in this case only the rise_time parameter is used. The fall_time parameter is ignored.

SEE ALSO

CxGetSAccelTimes, CxSetSAccelTimes, CxSetTrapAccelTimes, CxGetAccelTimesMin
CxGetTrapScrewAccel

Gets the current trapezoidal screw acceleration and deceleration

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetTrapScrewAccel(CxMechanism mech, double *accel,
                          double *decel)
```

ARGUMENTS

- **mech**: Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **accel**: The screw acceleration magnitude (deg/sec² or rad/sec²), depending on units. Default is degrees. See CxSetUnit in the Nodes, Frames, and Attributes section of the CODE API Programmer's Reference Manual for more information.
- **decel**: The screw deceleration magnitude (deg/sec² or rad/sec²), depending on units. Default is degrees. See CxSetUnit in the Nodes, Frames, and Attributes section of the CODE API Programmer's Reference Manual for more information.

DESCRIPTION

This function gets the current screw acceleration and deceleration magnitudes. When the acceleration type is CX_CONST_RAMP_ACCEL, these magnitudes determine the acceleration and deceleration during linear and circular interpolated moves if the rotational speed dominates the tool control frame (TCF) motion. If the current values are greater than the maximum screw acceleration and deceleration magnitudes, then the respective values will be set to these maximums.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxSetTrapScrewAccel, CxSetScrewAccelMax, CxGetScrewAccelMax
CxGetWeaveOnOff

Determines whether weaving is currently set to on or off

SYNOPSIS

```
#include <code/robpac.h>
long CxGetWeaveOnOff(CxMechanism mech, long *on_or_off)
```

ARGUMENTS

mech    Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.

on_or_off Returns the current weave setting.

DESCRIPTION

This function determines the current setting (on or off) for mechanism weaving. If weaving is enabled, the end effect or will oscillate in a specified pattern. This pattern is defined using the CxSetWeaveType function.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program gets the weave on or off setting.

```
#include <stdio.h>
#include <code/robpac.h>

void main( void )
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long on_or_off;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    /* Set weave type */
```
WARNINGS

Excessive use of weaving may damage the distal joints of the mechanism.

SEE ALSO

CxSetWeaveType, CxSetWeaveType, CXSetWeaveOnOff
CxGetWeaveType

Gets the weave type, weave amplitude and weave frequency

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetWeaveType(CxMechanism mech, long *type, double *ampl,
                    double *freq)
```

ARGUMENTS

- mech  Mechanism ID. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- type  Returns the currently defined weave type. See DESCRIPTION below for defined weaving types
- ampl  Returns the weaving pattern amplitude in the currently defined length units in the cell
- freq  Returns the currently defined weaving oscillation frequency

DESCRIPTION

This function gets the current setting for the weave pattern, the weave amplitude and the weave frequency of an oscillation pattern that is superimposed on the normal path motion of the tool TCF. Weaving is typically used in welding where the groove is large enough to require substantial filler material.

The available patterns are CX_SINE_WEAVE, CX_V_WEAVE, CX_Z_WEAVE, CX_DELTA_WEAVE, CX_CIRCLE_WEAVE, and CX_BOX_WEAVE, as shown in the following figure.

![Graphs of different weave types](image-url)
The weave amplitude that the user enters should be interpreted as one-half the total distance of oscillation of the tool. The frequency is the number of times the weave pattern will be repeated in one second.

Only mechanisms, which have terminal joints that are rotational and which intersect, can be used in weld weaving. CODE will return an error if the mechanism joints are not compatible for weld weaving. The trajectory rate used for the mechanism must be at least 4 times greater than the weave frequency; otherwise, it may not be possible to properly discriminate the weave pattern shape. For most practical 5- and 6-axis mechanisms, the weave frequency will probably lie in the range of 1-10 Hz.

The weld torch TCF should not be set too close to the joint axes, or the oscillation may cause the joint motion to exceed their speed limits. This will result in the mechanism slowing down. The user can use CODE simulation to determine whether this may occur. The user can use the desired amplitude, frequency, and the approximate radial distance of the weld torch TCF from the joint origin to calculate the required joint speed and then compare this value to those of the terminal joints.

Note that this API function only sets the weave parameters. It requires CxSetWeaveOnOff() to actually turn weaving on or off.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program gets the weave type, the weave amplitude and the weave frequency.

```c
#include <stdio.h>
#include <code/robpac.h>

void main( void )
{
    CxServer    Server;
```
CxMechanism mech;
CxNodeId s100;
long type;
double ampl;
double freq;

Server = CxOpenServer( "Testing", CX_SMEM, 0 ) ;
.
mech = CxOpenMechanism( Server, s100, CX_CONTROL ) ;

/* Set weave type */
CxGetWeaveType ( mech, &type, &ampl, &freq ) ;
.
CxRobpacExit ( ) ;
}

WARNINGS

Excessive use of weaving may damage the distal joints of the mechanism.

SEE ALSO

CxSetWeaveType, CxSetWeaveOnOff, CxGetWeaveOnOff
**CxICanMove**
Tests to see if a move is possible and then executes it if it is

**SYNOPSIS**
```c
#include <code/robpac.h>
long CxICanMove(CxMechanism mech, CxNodeId target, CxNodeId tcf, char axes[4], double a1, double a2, double a3, double x, double y, double z)
```

**ARGUMENTS**
- **mech** Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **Target** End node ID; This is the value returned by a call to CxGetNamedNodeId.
- **tcf** Node ID of tool to move. This is the value returned by a call to CxGetNamedNodeId.
- **axes** Indicates about which axes offset rotations apply.
- **a1** First rotation about the element frame (default in degrees).
- **a2** Second rotation about the element frame (default in degrees).
- **a3** Third rotation about the element frame (default in degrees).
- **x** Offset along the node’s x-axis.
- **y** Offset along the node’s y-axis.
- **z** Offset along the node’s z-axis.

**DESCRIPTION**
This function checks to determine if the target specified is reachable before executing the move. If the target is reachable, the move is executed and nothing is returned. If the target is not reachable, this function returns CX_FALSE and the move is not executed. The program then continues with the next move command in the queue.

When CxICanMove is invoked, it changes the motion interpolation type to CX_JOINT_INTERP and the blend policy to CX_MOVE_WAIT. After the function returns, the interpolation type and blend policy change back to the original settings for the remaining move sequences.

**RETURN VALUES**
This function returns CX_TRUE (1) if a move is possible, CX_FALSE (0) if it is not possible, or CX_ERROR (-1) if some other error occurs.

**ERRORS**
The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target does not exist.</td>
</tr>
<tr>
<td>CX_TCP_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see warnings).</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride for details.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxMoveRelNode
**CxLatchOnTrigger**
Sets the signal to trigger the position latching for a mechanism

**SYNOPSIS**
```c
#include <code/robpac.h>
#include <oac/oac.h>
#include <oac/oac_signal.h>
long CxLatchOnTrigger(CxServer Server, long latch_signal, CxMechanism mech, long trigger_flag)
```

**ARGUMENTS**
- **Server**
  Server ID
- **latch_signal**
  Signal to trigger the position latching operation. When using an MEI DSP card, this should be set to an unused MEI DSP or software signal.
- **mech**
  ID of mechanism on which position latching is to be performed.
- **trigger_flag**
  A flag indicating the condition which should trigger the position latching. This value is ignored when using an MEI DSP motion card (see Hardware and System Dependencies).

**DESCRIPTION**
This function sets up the trigger signal for position latching on a given mechanism. When this signal triggers based on the condition specified in trigger_flag, the axis positions are latched and stored. These values can be retrieved using CxGetLatchedAxis.

Not all trigger_flag values are necessarily supported, depending on the capabilities of the servo controller card (see Hardware and System Dependencies for details). In addition, the latch signal should reside on the same card used for controlling the mechanism (it is possible to use a software signal instead; however, this would only provide an approximation of the axis positions at the time the signal is triggered. Hence, its use is not recommended).

At present, CxLatchOnTrigger is only supported for use with the PMAC and MEI DSP cards.

**RETURN VALUES**
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**
The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
The following code segment shows how to set up a position latch operation. This example will latch the joint positions of the CxMechanism my_mech when SIG_1 changes its states form low to high.
CxServer my_server;
Long latch_signal;
CxMechanism my_mech;

/*my_mech has been opened for CX_CONTROL prior to this code segment */
latch_signal = SIG_1;
CxLatchOnTrigger(my_server, latch_signal, my_mech, LATCH_RISE_FLAG);

HARDWARE & SOFTWARE DEPENDENCIES

MEI DSP cards can only perform position latching based on I/O bit 22 (bit 6 of port 2); however, CIMServer requires that latch_signal must be valid. Thus, when using MEI DSP cards, this should be set to an unused software or MEI DSP signal. In addition, these cards can only latch when I/O bit 22 transitions from a high state (+5V) to a low state (0V); hence, the value of trigger_flag is ignored.

SEE ALSO

CxMoveRelNode
CxListConfig

Lists all configurations taught for a given mechanism

SYNOPSIS

```c
#include <code/robpac.h>
long CxListConfig(CxMechanism mech)
```

ARGUMENTS

- **mech**
  - Mechanism ID;
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION

This function lists all configurations taught for a given mechanism. The function uses CxGetNumConfig and CxGetConfigName to get all configurations taught for the given mechanism, then lists configuration names to stderr.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxGetNumConfig, CxGetConfigName, CxMoveToConfig, CxTeachConfig, CxDeleteConfig
**CxLoadMoveParameters**

Gets the trajectory motion parameters used in joint moves

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxLoadMoveParameters(CxMechanism mech, CxNodeId rob_id, CxTimedMove *move)
```

**ARGUMENTS**

- **mech**: Mechanism ID. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **rob_id**: Node ID of the mechanism.
- **move**: A data structure which is loaded with the mechanism's current motion parameters in the database.

**DESCRIPTION**

`CxLoadMoveParameters` is a convenience function which retrieves the current motion parameters from the server, then loads them into a data structure of type `CxTimedMove`. This data structure is defined as:

```c
typedef struct CxTimedMove
{
  long traj_accel_type; /* get from CxGetTrajAccelType */
  long accel_type; /* get from CxGetAccelType */
  double jnt_set; /* get from CxGetJointSpeed */
  double jnt_speed_max[CX_MAXJOINTS]; /* get from CxGetMaxJntSpeed */
  double jnt_accel_max[CX_MAXJOINTS]; /* get from CxGetJntAccelMax */
  double jnt_decel_max[CX_MAXJOINTS]; /* get from CxGetJntAccelDecel */
  double jnt_accel[CX_MAXJOINTS]; /* get from CxGetJntAccel */
  double jnt_decel[CX_MAXJOINTS]; /* get from CxGetJntAccelDecel */
  double rise_S_time; /* get from CxGetSAccelTimes */
  double fall_S_time; /* get from CxGetSAccelTimes */
  double rise_time_min; /* get from CxGetAccelTimesMin */
  double fall_time_min; /* get from CxGetAccelTimesMin */
  double rise_time; /* get from CxGetTrapAccelTimes */
  double fall_time; /* get from CxGetTrapAccelTimes */
} CxTimedMove;
```

The user can also load the data structure members directly, but must ensure that the units are consistent with those currently in use (degrees or radians, and the current length units as reported by `CxSetUnit`). The default units are millimeters and degrees.

This function applies only to joint moves and not to curvilinear type moves. It is intended to be used in conjunction with the functions `CxGetJointMoveTime` and `CxSetJointMoveTime`. The programmer can use these functions to get and set the times required to perform certain joint moves, but should be aware that the trajectory calculations in the server are governed by the current motion parameters. To achieve the desired time, the user must ensure that the motion parameters used in the move time calculations are the same as those in the server database. The user can change these settings by using the "get" counterparts of the "get" functions mentioned in the comments of the data structure definition.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

The following code will get the motion parameters for a mechanism:

```cpp
CxMechanism mech;
CxTimedMove move;
CxNodeId rob_id;

CxLoadMoveParameters(mech, rob_id, &move);
```

SEE ALSO

`CxSetJointMoveTime`, `CxGetJointMoveTime`
**CxMoveAllAxes**

Moves all independent joints to the specified values

**SYNOPSIS**

```c
#include <code/robpac.h>)
long CxMoveAllAxes (CxMechanism mech, double *axes_values)
```

**ARGUMENTS**

- `mech` Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `axes_values` This is an array of desired values for all joints of the mechanism. These values will be in degrees or radians for rotational joints. See CxSetUnit in the Nodes, Frames and Attributes section of the CODE API Programmer’s Reference Manual- Volume 2 for more information.

**DESCRIPTION**

This function causes all of a mechanism’s independent joints to move to the specified absolute joint values. Allowable motion interpolation types are CX_JOINT_INTERP and CX_LINEAR_INTERP.

**NOTE:** For purely serial mechanisms, joint numbers begin with zero and increment along the mechanism linkage. All of the independent joints are numbered first, followed by all of the linearly dependent joints and, finally, all of the functionally dependent joints. For example, if a mechanism had six independent joints, two linearly dependent joints, and one functionally dependent joint, the independent joints would be numbered 0 through 5 in serial order, the linear joints would be 6 and 7, and the functional joint would be 8. Numbers are independent of any names you may assign in CODE.

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The attempted move would require one or more joints to move outside of the joint limits, either at the end point or somewhere along the motion path. The motion path is determined by the current motion type setting.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_OUT_OF_RANGE</td>
<td>The attempted move would require the specified joint to move outside the joint's limits, either at the end point or somewhere along the motion path. The motion path is determined by the current motion_type setting.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

This function will cause the mechanism to move under CIMControl. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode.

**EXAMPLE**

The following code will move all six axes of a six degree-of-freedom mechanism to zero.

```c
CxMechanism mech;
double axes_values[CX_MAXJOINTS];
long i;
for(i=0; i<6; i++)
    axes_value[i] = 0.0;
CxMoveAllAxes(mech, axes_value);
```

**SEE ALSO**

CxSetTipSpeed, CxSetInterpType, CxSetJointSpeed, CxMoveSingleAxis
**CxMoveAway**

Moves the specified tool from its current location along its Z axis

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxMoveAway(CxMechanism mech, CxNodeId tool, double z)
```

**ARGUMENTS**

- **mech**: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **tool**: Node ID of a tree node attached to the last joint of the current mechanism, whose frame will be used to determine the move. This is the value returned by a call to CxGetNamedNodeId.
- **z**: Distance tool is moved along its Z-axis from its current pose.

**DESCRIPTION**

This function provides a convenient method to back a tool away from a placed part. In order to use this function, the workcell modeler must adhere to the convention of using a tool’s Z axis as the vector along which the tool normally moves. If this is not the case, or if the tool needs to be moved along an arbitrary vector, use the more general function CxMoveRelNode.

Note that in order to move away from an object, it is often necessary to supply a negative argument as the Z_dist.

Calling CxMoveAway is equivalent to calling CxMoveRelNode with the following arguments:

```c
CxMoveRelNode(mech,tool,tool,"XYZ",0.,0.,0.,0.,0., z);
```

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The specified distance would cause the mechanism to move outside its envelope, either at the end-point or somewhere along the motion path. The motion path is determined by the current motion type setting.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>Specified tool does not exist, or is not attached to the last joint of the current mechanism.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
</tbody>
</table>
### Error Code Description

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

### WARNINGS

When the CIMServer is set for runtime, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

Note that in order to move away from an object, it is often necessary to supply a negative argument as the Z_dist.

### EXAMPLE

The following code fragment defines a reusable function for placing objects. The user specifies an object to be placed, a location for placing the object (defined as a node frame), a tool to use, an approach and retraction distance, and a signal and value for actuating the gripper.

```c
place_object (CxServer Server, CxMechanism mech, CxNodeId object,
             CxNodeId location, CxNodeId tool, double z_dist,
             long signal, long value)
{
    CxMoveNearNode (mech, location, tool, z_dist);
    CxMoveToNode (mech, location, tool);
    CxSetSignal (Server, signal, value);
    attach_node (Server, object, location);
    CxMoveAway (mech, tool, -z_dist);
}
```

### SEE ALSO

CxMoveRelNode, CxSetTipSpeed, CxSetInterpType, CxSetJointSpeed
**CxMoveNearNode**

Moves the tool near a node, offset along the target’s Z axis

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxMoveNearNode(CxMechanism mech, CxNodeId target, CxNodeId tool, double z)
```

**ARGUMENTS**

- `mech`: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `target`: Tree node to move to; This is the value returned by a call to `CxGetNamedNodeId`.
- `tool`: Tool to move; This is the value returned by a call to `CxGetNamedNodeId`.
- `z`: Tool final offset from the node along target z-axis.

**DESCRIPTION**

This function provides a convenient method to move the mechanism’s tool relative to an element or node part offset along the target’s Z-axis. In order to use this function, the workcell modeler must adhere to the convention of using a target’s Z-axis as the vector along which the tool normally moves. If this is not the case, or if the tool needs to be moved along an arbitrary vector, the more general function `CxMoveRelNode` should be used.

Calling `CxMoveNearNode` is equivalent to calling `CxMoveRelNode` with the following arguments.

`CxMoveRelNode(mech,target,tool,"XYZ",0.,0.,0.,0.,0.,z);`

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 if an error occurs; and 1 if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Messages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target node does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the current mechanism’s last joint.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach either at the final pose or at some point along the path from the initial mechanism pose and the final pose. The motion path is determined by the current motion type.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See <code>CxSetTrajectoryOverride</code> for details.</td>
</tr>
<tr>
<td>Error Messages</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

**SEE ALSO**

CxMoveRelNode, CxSetInterpType
**CxMoveRelArc**

Moves the tool along an arc defined by two frames and path axis

**SYNOPSIS**

```c
define CxMoveRelArc(CxMechanism mech, CxNodeId target, CxNodeId tcf, long path_axis, char axes[4], double a1, double a2, double a3, double x, double y, double z)
```

**ARGUMENTS**

- **mech** Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **target** Node ID of node that completes the arc. This is the value returned by a call to CxGetNamedNodeID.
- **tcf** Node ID of tool to move along arc. This is the value returned by a call to CxGetNamedNodeID.
- **path_axis** Tool axis arc tangent direction (0=x, 1=y, 2=z).
- **axes** Indicates about which axes offset rotations apply.
- **a1** First rotation about the target frame (in degrees by default).
- **a2** Second rotation about the target frame (in degrees by default).
- **a3** Third rotation about the target frame (in degrees by default).
- **x** Offset along the node's x-axis.
- **y** Offset along the node's y-axis.
- **z** Offset along the node's z-axis.

**DESCRIPTION**

This function directs the mechanism to move along a circular arc in space from the current tool frame origin to the target frame origin and tangent to the path_axis of the current tool TCF frame.

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach either at the final pose or at some point along the path from the initial mechanism pose and the final pose.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see WARNINGS).</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Path_axis is not 0, 1, or 2.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_UNKNOWN_CURVE_TYPE</td>
<td>Cannot decide curve type of the arc.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter axes, a four character array, can take only the values x, y, z, X, Y, and Z. The string must be null-terminated and axes must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (a1, a2, a3).

The following are all legal:

- "XYZ"
- "ZZZ"
- "yzx"

The following are all illegal, for the indicated reasons:

- "ZY"   Fewer than three letters passed
- "zzz"  More than three letters passed
- "ZCy"  Illegal character (C)

**SEE ALSO**

CxSetTipSpeed, CxSetToolMotionType, CxMoveRelArc3
**CxMoveRelArc3**

Moves a tool along an arc defined by two frames and an intermediate frame

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxMoveRelArc3(CxMechanism mech, CxNodeId target, CxNodeId tcf,
                    CxNodeId mid_arc, char axes[4], double a1, double a2, double a3,
                    double x, double y, double z)
```

**ARGUMENTS**

- `mech`: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `Target`: Node ID of node that completes the arc. This is the value returned by a call to `CxGetNamedNodeID`.
- `tcf`: Node ID of tool to move along arc. This is the value returned by a call to `CxGetNamedNodeID`.
- `mid_arc`: Node ID of intermediate frame defining three point arc. This is the value returned by a call to `CxGetNamedNodeID`.
- `axes`: Indicates about which axes offset rotations apply.
- `a1`: First rotation about the target frame (default in degrees).
- `a2`: Second rotation about the target frame (rotation in degrees).
- `a3`: Third rotation about the target frame (default in degrees)
- `x`: Offset along the node's x-axis.
- `y`: Offset along the node's y-axis.
- `z`: Offset along the node's z-axis.

**DESCRIPTION**

This function directs the mechanism to move along a circular arc in space from the current tool frame through the `mid_arc` frame to the `target` frame.

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>The specified <code>mid_arc</code> node does not exist.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach either at the final pose or at some point along the path from the initial mechanism pose and the final pose.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see WARNINGS).</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_UNKNOWN_CURVE_TYPE</td>
<td>Cannot decide curve type of arc.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program moves the TCF along the arc created by three points, i.e., frames of table, path and TCF.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100, tcf, table;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "s100",  &s100 );
    CxGetNamedNodeId( Server, "tcf",  &tcf );
    CxGetNamedNodeId( Server, "table",  &table );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    
    CxMoveRelArc3( mech, table, tcf, path, "XYZ", 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 );
    
    CxRobpacExit();
}
```

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is *strongly* recommended that all programs be tested in simulation before being
executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter \texttt{axes}, a four character array, can take only the values \texttt{x}, \texttt{y}, \texttt{z}, \texttt{X}, \texttt{Y}, and \texttt{Z}. The string must be null-terminated and \texttt{axes} must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (\texttt{a1}, \texttt{a2}, \texttt{a3}).

The following are all legal:

- "XYZ"
- "ZZZ"
- "yZx"

The following are all illegal, for the indicated reasons:

- "ZY" Fewer than three letters passed
- "zzzz" More than three letters passed
- "ZCy" Illegal character (C)

\textbf{SEE ALSO}

\texttt{CxSetTipSpeed, CxSetScrewSpeed, CxSetToolMotionType}
CxMoveRelArcNormal
Moves along an arc defined by the TCF, arc center and normal to arc plane

SYNOPSIS
#include <code/robpac.h>
long CxMoveRelArcNormal(CxMechanism mech, CxNodeId target, CxNodeId
tcf, CxVector arc_center, CxVector arc_normal, double arc_angle)

ARGUMENTS
mech Mechanism ID of mechanism open for control. This is a CxMechanism ID returned
by a call to CxOpenMechanism.
target Node ID of node that completes the arc. This is the value returned by a call to
CxGetNamedNodeId.
tcf Node Id of tool to move along arc. This is the value returned by a call to
CxGetNamedNodeId.
arc_center Point on the line, normal to the arc and through the arc center.
arc_normal Unit vector, which is normal to the arc, plane.
arc_angle Rotation angle about arc_normal (default in degrees).

DESCRIPTION
This function directs the mechanism to move along a circular arc in space from the current tool frame origin
about the arc_normal unit vector, which is normal to the arc plane and which passes through
arc_center. The arc_angle can be positive or negative and can also be greater than 360 degrees. The
TCF z-axis must be parallel to the arc_normal vector or the move will return an error. The reference
node (target) provides the reference frame for the vectors arc_center and arc_normal. The tool
motion type is forced to CX_FIXED_ORIENT for this move.

RETURN VALUES
This function returns 0 (CX_OK) if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1
(CX_GOT_SIGNAL) if a signal interrupt occurs.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_ARC_NORMAL_REF_NODE_NOT_FOUND</td>
<td>The reference node, which is used to define the arc center and normal parameters, is not found.</td>
</tr>
<tr>
<td>CX_CANT_TRACK_ARC_NORMAL_MOVE</td>
<td>Motion tracking should not be turned on for this type of move.</td>
</tr>
<tr>
<td>Error Codes</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_ARC_NORMAL_INVALID_PARAM</td>
<td>An arc parameter value is not entered correctly.</td>
</tr>
<tr>
<td>CX_TOOL_NOT_NORMAL_TO_ARC</td>
<td>The TCF z-axis is not normal to the arc plane.</td>
</tr>
<tr>
<td>CX_ARC_NORMAL_RADIUS_ZERO</td>
<td>The tcf z-axis lies on the arc center.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

**EXAMPLE**

The following program moves the TCF 375 degrees about the normal vector, which is referenced, to node door_arc.

```c
#include <stdio.h>
#include <code/robpac.h>

void main ( void )
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId   s100, tcf, door_arc;
    CxVector arc_center, arc_normal;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId ( Server, "s100", &s100 ) ;
    CxGetNamedNodeId ( Server, "tcf", &tcf ) ;
    CxGetNamedNodeId ( Server, "door_arc", &door_arc ) ;
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;

    arc_center[0] = 125.6; arc_center[1] = -23.4; arc_center[2] = 0.0;
    arc_normal[0] = 0.0, arc_normal[1] = 0.0; arc_normal[2] = 1.0;
    CxMoveRelArcNormal ( mech, door_arc, tcf, arc_center, arc_normal, 375.0 ) ;

    CxRobpacExit();
}
```
SEE ALSO

CxSetTipSpeed, CxSetToolMotionType, CxMoveRelArc3
CXMoveRelNode

Moves the tool relative to a node with an arbitrary offset

SYNOPSIS

`#include <code/robpac.h>
long CXMoveRelNode(CxMechanism mech, CxNodeId target, CxNodeId tcf, char axes[4], double a1, double a2, double a3, double x, double y, double z)`

ARGUMENTS

- `mech`: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CXOpenMechanism.
- `target`: Node ID of node to move relative to. This is the value returned by a call to CXGetNamedNodeId.
- `tcf`: Node ID of tool to move. This is the value returned by a call to CXGetNamedNodeId.
- `axes`: Indicates about which axes offset rotations apply
- `a1`: First rotation about the element frame (in degrees, by default)
- `a2`: Second rotation about the element frame (in degrees by default)
- `a3`: Third rotation about the element frame (in degrees by default)
- `x`: Offset along the node’s x-axis
- `y`: Offset along the node’s y-axis
- `z`: Offset along the node’s z-axis

DESCRIPTION

This function is the most general of the inverse kinematics move commands. It moves a TCF node to a target node at a given offset. Most other inverse kinematic moves are derivatives of CXMoveRelNode.

RETURN VALUES

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CXGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach either at the final pose or at some point along the path from the</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>initial mechanism pose and the final pose.</td>
<td></td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see warnings).</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program tries to chase the moving object. It sets the motion tracking to CX_ON. Then it moves the TCF to the moving object.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
  CxServer    Server;
  CxMechanism mech;
  CxNodeId   s100, moving_target, tcf;
  long      is_still_tracking;

  Server = CxOpenServer( "Testing", CX_SMEM, 0 );
  .
  mech = CxOpenMechanism( Server, s100, CX_CONTROL );
  CxSetBlendPolicy( mech, CX_MOVE_WAIT);
  .
  /* start the target in motion */
  /* Chase moving object */
  CxSetMotionTrack( mech, CX_ON );
  CxMoveRelNode( mech, moving_target, tcf, "XYZ", 0.0, 0.0, 0.0, 5.0, 5.0, 5.0 );
  .
  CxRobpacExit();
}
```

**WARNINGS**

When the CIMServer is set for runtime, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being
executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter axes, a four character array, can take only the values x, y, z, X, Y, and Z. The string must be null-terminated and axes must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (a1, a2, a3).

The following are all legal:

"XYZ"
"ZZZ"
"y2x"

The following are all illegal, for the indicated reasons:

"ZY" Fewer than three letters passed
"zzzz" More than three letters passed
"ZCy" Illegal character (C)

SEE ALSO

cxSetTipSpeed, cxSetInterpType, cxSetToolMotionType, cxSetScrewSpeed, cxSetJointSpeed
**CxMoveRelPath**

Moves the tool with an arbitrary offset along a path of linked segments

**SYNOPSIS**

```
#include <code/robpac.h>
long CxMoveRelPath(CxMechanism mech, CxNodeId target, CxNodeId tool, 
                  char start_seg[CX_MAXNAME], end_seg[CX_MAXNAME], char axes[4], 
                  double a1, double a2, double a3, double x, double y, double z)
```

**ARGUMENTS**

- **mech**
  - Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

- **target**
  - Node ID of node containing path along which you are moving. This is the value returned by a call to CxGetNamedNodeId.

- **tool**
  - Node ID of tool to move. This is the value returned by a call to CxGetNamedNodeId.

- **start_seg**
  - Name of starting segment in curve.

- **end_seg**
  - Name of ending segment in curve.

- **axes**
  - Indicates about which axes offset rotations apply.

- **a1**
  - First rotation about the element frame (in degrees, by default).

- **a2**
  - Second rotation about the element frame (in degrees, by default).

- **a3**
  - Third rotation about the element frame (in degrees, by default).

- **x**
  - Offset along the node’s x-axis.

- **y**
  - Offset along the node’s y-axis.

- **z**
  - Offset along the node’s z-axis.

**DESCRIPTION**

This function tells the mechanism to move along a linked list of curve segments of varying types (LINEAR, CIRCULAR, and LINCIRC) by one function call.

Each segment in a curve has a name, which may be used as the arguments for the parameters `start_seg` and `end_seg`. These parameters must be entered as a character string. If `start_seg` is entered as "" (the empty string), then the first segment in the linked list is first segment in the path and all segments between the first segment and `end_seg` will be traversed. If `end_seg` is entered as "" then `end_seg` is assumed to be the final segment in the path and all segments from `start_seg` to the final segment are traversed. If both `start_seg` and `end_seg` are " ", then the mechanism will move along the entire path; otherwise, the mechanism will move from `start_seg` to, but not including `end_seg`.

If `start_seg` is the same as `end_seg`, then the mechanism will move along one segment named `start_seg`.

Note that the mechanism can move either forward or backward along a series of path segments, depending upon the order in which `start_seg` and `end_seg` are entered.

The parameters `axes`, `a1`, `a2`, `a3`, `x`, `y`, and `z` define the offset pose of the tool with respect to the path during the move.
RETURN VALUES
This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target curve does not exist (or segments do not exist).</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach either at the final pose or at some point along the path from the start_seg to the end_seg.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see WARNINGS).</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_MISMATCHED_GEOM_PATH</td>
<td>Geometry is not of type CX_CURVE, CX_SWEPTT, or CX_SWEPTR.</td>
</tr>
<tr>
<td>CX_INVALID_START_FRAME</td>
<td>Specified start frame does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_END_FRAME</td>
<td>Specified ending frame does not exist</td>
</tr>
<tr>
<td>CX_PATH_SEG_INTERP_ERROR</td>
<td>Start segment cannot be same as end segment.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program creates a 3-D path and then moves the TCF relative to that path.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
```
CxNodeId s100, path, tcf;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxGetNamedNodeId( Server, "s100", &s100 );
CxGetNamedNodeId( Server, "path", &path );
CxGetNamedNodeId( Server, "tcf", &tcf );
mech = CxOpenMechanism( Server, s100, CX_CONTROL );

/* Create a curve to be used for the tool movement */
CxAddCurveSeg( Server, path, "", "pt1", "XYZ", 0.0, 0.0, 0.0,
0.0, 0.0, 0.0 );
CxAddCurveSeg( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
0.0, 140.0, 500.0 );
CxAddCurveSeg( Server, path, "pt2", "pt3", "XYZ", 0.0, 0.0, 0.0,
40.0, 560.0, 700.0 );
CxAddCurveSeg( Server, path, "pt3", "pt4", "XYZ", 0.0, 0.0, 0.0,
-1800.0, -1230.0, 300.0 );
CxAddCurveSeg( Server, path, "pt4", "pt5", "XYZ", 0.0, 0.0, 0.0,
100.0, -610.0, 1100.0 );
CxAddCurveSeg( Server, path, "pt5", "pt6", "XYZ", 0.0, 0.0, 0.0,
-600.0, -1000.0, 0.0 );

/* Move along the path */
CxMoveRelPath( mech, path, tcf, "", "", "XYZ", 0.0, 0.0, 0.0,
5.0, 5.0, 5.0 );

. .
CxRobpacExit();
}

WARNINGS

When the CIMServer is set for runtime, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter axes, a four character array, can take only the values x, y, z, X, Y, and Z. The string must be null-terminated. This parameter must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (a1, a2, a3).

The following are all legal:
"XYZ"
"ZZZ"
"yZx"

The following are all illegal, for the indicated reasons:
"ZY" Fewer than three letters passed
"zzzz" More than three letters passed
"zCy" Illegal character (C)

SEE ALSO

CxSetTipSpeed, CxSetToolMotionType, CxMoveRelPathFrame, CxSetBlendPolicy
CxMoveRelPathFrame

Moves the tool with an arbitrary offset relative to a segment on a path

SYNOPSIS

```c
#include <code/robpac.h>
long CxMoveRelPathFrame(CxMechanism mech, CxNodeId path, CxNodeId tool,
                        char path_seg[CX_MAXNAME], char axes[4], double a1, double a2,
                        double a3, double x, double y, double z)
```

ARGUMENTS

- **mech**: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **path**: Node ID of node containing path segment. This is the value returned by a call to `CxGetNamedNodeId`.
- **tool**: Node ID of tool to move. This is the value returned by a call to `CxGetNamedNodeId`.
- **path_seg**: Name of path segment in curve to move relative to.
- **axes**: Indicates about which axes offset rotations apply.
- **a1**: First rotation about the segment frame (in degrees, by default).
- **a2**: Second rotation about the segment frame (in degrees, by default).
- **a3**: Third rotation about the segment frame (in degrees, by default).
- **x**: Offset along the node’s x-axis.
- **y**: Offset along the node’s y-axis.
- **z**: Offset along the node’s z-axis.

DESCRIPTION

This function tells the mechanism to move relative to a named path segment frame. Each segment in a curve has a name, which may be used as the argument for `path_seg`. This parameter must be entered as a character string. If `path_seg` is entered as "" (the empty string), then the first segment in the linked list is the target segment frame. The parameters `axes`, `a1`, `a2`, `a3`, `x`, `y`, and `z` pose the offset of the tool with respect to the path segment frame.

RETURN VALUES

This function returns 0 if the move completes successfully; -1 `(CX_ERROR)` if an error occurs; and 1 `(CX_GOT_SIGNAL)` if a signal interrupt occurs.

ERRORS

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function.

The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
</table>

...
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>The specified target curve does not exist (or segments do not exist).</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>The specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>The move is outside the mechanism’s reach.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string (see WARNINGS).</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program creates the path and moves the TCF relative to the "pt4" segment of the path.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100, path, tcf, table;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNode( Server, "s100", &s100 );
    CxGetNamedNode( Server, "path", &path );
    CxGetNamedNode( Server, "tcf", &tcf );
    CxGetNamedNode( Server, "table", &table );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );

    /* Create a curve to be used for the tool movement */
    CxAddCurveSeg( Server, path, ",", "pt1", "XYZ", 0.0, 0.0, 0.0,
                  0.0, 0.0, 0.0 );
    CxAddCurveSeg( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
                  0.0, 140.0, 500.0 );
    CxAddCurveSeg( Server, path, "pt2", "pt3", "XYZ", 0.0, 0.0, 0.0,
                  40.0, 560.0, 700.0 );
    CxAddCurveSeg( Server, path, "pt3", "pt4", "XYZ", 0.0, 0.0, 0.0
                  -1800.0, -1230.0, 300.0 );
```
WARNINGS

When the CIMServer is set for runtime, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter axes, a four character array, can take only the values x, y, z, X, Y, and Z. The string must be null-terminated. This parameter must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (a1, a2, a3).

The following are all legal:

"XYZ"
"ZZZ"
"yZx"

The following are all illegal, for the indicated reasons:

"ZY" Fewer than three letters passed
"zzzz" More than three letters passed
"ZCy" Illegal character (C)

SEE ALSO

CxSetTipSpeed, CxSetInterpType, CxSetJointSpeed, CxMoveRelPath
**CxMoveRelTool**

Moves a tool relative to its current pose

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxMoveRelTool(CxMechanism mech, CxNodeId tool, char axes[4],
    double a1, double a2, double a3, double x, double y, double z)
```

**ARGUMENTS**

- `mech`  
  Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `tool`  
  Node ID of tool to move. This is the value returned by a call to `CxGetNamedNodeId`.
- `axes`  
  Indicates about which axes offset rotations apply.
- `a1`  
  First offset rotation about the tool frame (in degrees, by default).
- `a2`  
  Second offset rotation about the tool frame (in degrees, by default).
- `a3`  
  Third offset rotation about the tool frame (in degrees, by default).
- `x`  
  Offset along the node's x-axis.
- `y`  
  Offset along the node's y-axis.
- `z`  
  Offset along the node's z-axis.

**DESCRIPTION**

This function provides a convenient method to move a tool relative to itself. Calling `CxMoveRelTool` is equivalent to calling `CxMoveRelNode` with these arguments: `CxMoveRelNode(mech, tool, tool, axes, a1, a2, a3, x, y, z);`

**RETURN VALUES**

This function returns **0** if the move completes successfully; **-1** (**CX_ERROR**) if an error occurs; and **1** (**CX_GOT_SIGNAL**) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TCP_NOT_FOUND</td>
<td>Specified tool does not exist or is not attached to the mechanism's last joint.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>Move is outside the mechanism's reach at the final pose or at some point along the path from the initial mechanism pose and the final pose. The motion path is determined by the current motion type.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See <code>CxSetTrajectoryOverride</code> in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axis string.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is *strongly* recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

The parameter `axes`, a four character array, can take only the values x, y, z, X, Y, and Z. The string must be null-terminated. This parameter must be given three characters. If fewer than three rotations are desired, then zero (0) must be passed as one or more of the rotation parameters (a1, a2, a3).

The following are all legal:

- "XYZ"
- "ZZZ"
- "yZx"

The following are all illegal, for the indicated reasons:

- "ZY" Fewer than three letters passed
- "zzzz" More than three letters passed
- "ZCy" Illegal character (C)

**SEE ALSO**

CxMoveRelNode, CxMoveAway, CxSetTipSpeed, CxSetInterpType, CxSetJointSpeed
**CxMoveSingleAxis**

Moves a single joint to the specified value

**SYNOPSIS**

```cpp
#include <code/robpac.h>
long CxMoveSingleAxis(CxMechanism mech, long axis, double axis_value)
```

**ARGUMENT**

- `mech`: Mechanism ID of mechanism open for control.
  This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `axis`: Joint number to be moved.
- `axis_value`: Absolute position of the joint after move. A rotational joint's value is defined in degrees or radians (see `CxSetUnit` in the `Nodes, Frames and Attributes` section of the `CODE API Programmer's Reference Manual- Volume 2` for more information). A translational joint's value is defined in linear units.

**DESCRIPTION**

This function causes a single axis to move to a specified absolute position. The specified axis must be independent.

**NOTE:** For purely serial mechanisms, joint numbers begin with zero and increment along the mechanism linkage. All of the independent joints are numbered first, followed by all of the linearly dependent joints and, finally, all of the functionally dependent joints. For example, if a mechanism had six independent joints, two linearly dependent joints, and one functionally dependent joint, the independent joints would be numbered 0 through 5 in serial order, the linear joints would be 6 and 7, and the functional joint would be 8. Numbers are independent of any names you may assign in the CIMServer workcell model.

For non-serial mechanisms, use the `CxGetJntNumber` to get the joint number of a named joint of a given mechanism.

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_OUT_OF_RANGE</td>
<td>The attempted move would require the specified joint to move outside the joint’s limits, either at the end point or somewhere along the motion path. The motion path is determined by the current motion_type setting.</td>
</tr>
<tr>
<td>CX_INVALID_JOINT_NUMBER</td>
<td>Specified joint number is not valid.</td>
</tr>
</tbody>
</table>
### Error Code Description

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See CxSetTrajectoryOverride in the Process and Mechanism Control library for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

### EXAMPLE

The following program opens a mechanism, sets the blend policy to CX_MOVE_TO and issues commands to move joints. It moves axis #0 to a joint value of 100.0 and axis #1 to a joint value of 120.0. Since the blend policy is set to CX_MOVE_TO, function calls will return even if the corresponding motions are incomplete. To avoid calling function CxRobpacExit before all the motions are completed, CxWaitForEndOfMotion is called.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO);

    CxMoveSingleAxis( mech, 0, 100.0 );
    CxMoveSingleAxis( mech, 1, 120.0 );

    CxWaitForEndOfMotion( mech );
    CxRobpacExit();
}
```

### WARNINGS

When the CIMServer is set for runtime, this function will cause the mechanism to move. To avoid unnecessary collisions, it is strongly recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

### SEE ALSO

CxMoveAllAxes, CxGetJointSpeed, CxSetJointSpeed
CxMoveToConfig
Moves to a named configuration

SYNOPSIS
#include <code/robpac.h>
long CxMoveToConfig(CxMechanism mech, char *config_name)

ARGUMENTS
mech Mechanism ID of mechanism open for control.
This is a CxMechanism ID returned by a call to CxOpenMechanism.
config_name Configuration name.

DESCRIPTION
This function moves to a named configuration, which contains a set of joint values associated with a
mechanism. The configuration is taught using the CxTeachConfig.
If the interpolation type (CxSetInterpType) is CX_LINEAR_INTERP the mechanism will move in a
straight line to the configuration using the last joint frame as the TCF.

RETURN VALUES
This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1
(CX_GOT_SIGNAL) if a signal interrupt occurs.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorMessage
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_CONFIG_NOT_EXIST</td>
<td>Named configuration does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>Move is outside the mechanism’s reach.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program first creates a 3-D path, then saves the current configuration. Next, it moves the TCF
along the path. It then moves the TCF back to the original configuration. Finally, it deletes the configuration.

#include <stdio.h>
#include <code/robpac.h>

void main(void)
{

CxServer    Server;
CxMechanism mech;
CxNodeId   s100, path, tcf;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxGetNamedNodeId( Server, "s100", &s100 );
CxGetNamedNodeId( Server, "path", &path );
CxGetNamedNodeId( Server, "tcf", &tcf );
mech = CxOpenMechanism( Server, s100, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO);

/* Create a curve to be used for the tool movement */
CxAddCurveSeg( Server, path, ",", "pt1", "XYZ", 0.0, 0.0, 0.0,
               0.0, 0.0, 0.0 );
CxAddCurveSeg( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
               0.0, 140.0, 500.0 );
.
.
CxTeachConfig( mech, "orig_config", , CX_ANIMATION );

CxMoveRelPath( mech, path, tcf, ",", ",", "XYZ", 0.0, 0.0, 0.0,
               5.0, 5.0, 5.0 );

CxMoveToConfig( mech, "orig_config" );

CxDeleteConfig( mech, "orig_config" );
.
.
CxRobpacExit();
}

SEE ALSO

CxDeleteConfig, CxTeachConfig, CxGetConfigName CxGetNumConfig, CxListConfig, CxSetInterpType
**CxMoveToNode**

Moves a tool to the specified node with no offset

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxMoveToNode(CxMechanism mech, CxNodeId target, CxNodeId tcf)
```

**ARGUMENTS**

- `mech` Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `target` Node ID of node to move to. This is the value returned by a call to `CxGetNamedNodeId`.
- `tcf` Node ID of tool to use. This is the value returned by a call to `CxGetNamedNodeId`.

**DESCRIPTION**

The function `CxMoveToNode` is used to move a tool attached to a mechanism such that its final pose coincides with the given target node. The mechanism will move from its current location to the specified pose at the current speed setting, using the current interpolation type.

Calling `CxMoveToNode` is equivalent to calling `CxMoveRelNode` with these arguments:

```c
CxMoveRelNode(mech, target, tcf,"XYZ",0.,0.,0.,0.,0.,0.);
```

**RETURN VALUES**

This function returns 0 if the move completes successfully; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if a signal interrupt occurs.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_TARGET_NOT_FOUND</td>
<td>Specified target node does not exist.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>Specified tool does not exist or is not attached to the mechanism.</td>
</tr>
<tr>
<td>CX_TARGET_OUT_OF_REACH</td>
<td>Move is outside the mechanism’s reach either at the final pose or at some point along the path from the initial mechanism pose and the final pose. The motion path is determined by the current motion type.</td>
</tr>
<tr>
<td>CX_TRAJECTORY_OVERRIDE_NOT_SET</td>
<td>Trajectory override flag is set to CX_FALSE. See <code>CxSetTrajectoryOverride</code> in the Controller Management section of the CODE API Programmer's Reference Manual - Volume 1 for details.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CX_TRAJECTORY_ERROR</td>
<td>Tclass number is invalid.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

**WARNINGS**

When the CIMServer is set for run time, this function will cause the mechanism to move. To avoid unnecessary collisions, it is *strongly* recommended that all programs be tested in simulation before being executed in run-time mode. Also, care must be taken to ensure that the actual workcell and its model are in the same state when execution begins.

Since this routine causes the mechanism to move its tool directly to the part frame, there is often a danger of collision. This is avoided by first calling CxMoveNearNode or CxMoveRelNode to pose the tool at a safe approach to the part and then calling CxMoveToNode (see **EXAMPLE**).

**EXAMPLE**

The following call will cause the current mechanism to move a tool identified as Gripper such that its control frame is coincident (lined up) with the node Peg.

```cpp
CxMoveToNode(mech, Peg, Gripper);
```

In this next example, the mechanism will move to a set of four pegs in a loop. The mechanism first moves to a safe location 100 units above the pegs, then moves to the pegs, and finally backs away.

```cpp
CxMechanism mech;
CxNodeId pegs[4];
CxNodeId gripper;

for(i=0; i<4; i++) {
    CxMoveNearNode(mech, pegs[i], Gripper, 100.0);
    CxMoveToNode(mech, pegs[i], Gripper);
    CxMoveAway(mech, Gripper, 100.0);
}
```

**SEE ALSO**

CxMoveRelNode, CxSetTipSpeed, CxSetInterpType, CxSetToolMotionType, CxSetScrewSpeed, CxSetJointSpeed
CxReteachNode
Changes the pose of an existing node to the current TCF pose

SYNOPSIS
#include <code/robpac.h>
long CxReteachNode(CxMechanism mech, CxNodeId node, CxNodeId tcf)

ARGUMENTS
mech Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
node Node ID of node of interest. This is the value returned by a call to CxGetNamedNodeId.
tcf Node ID of the TCF node of the mechanism. This is the value returned by a call to CxGetNamedNodeId.

DESCRIPTION
This function is used to set a node’s pose with respect to its parent so that the node will coincide (line up) with the TCF node.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Invalid node ID.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>Invalid TCF node ID.</td>
</tr>
<tr>
<td>CX_MACHINE_OUT_OF_MEMORY</td>
<td>Memory allocation failed.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_INVALID_NODE_ID</td>
<td>The node number does not match the given node ID number.</td>
</tr>
<tr>
<td>CX_NODE_IS_CUT_OUT</td>
<td>Given node has been cut out.</td>
</tr>
<tr>
<td>CX_MECH_IS_NOT_OPENED_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following segment moves the TCF to a node called feature, moves away 200 units, then creates a node named approach_node as a child of the world. The approach_node is then re-taught using the CxReteachNode API function.
CxServer test_server;
CxMechanism mech;
CxNodeId feature, tcf, world, approach_node;

CxMoveToNode( mech, feature, tcf );
CxMoveAway( mech, tcf, 200.0 );
CxTeachNode( mech, "approach_node", world, tcf );
CxGetNamedNodeId( test_server, "approach_node", &approach_node );

CxReteachNode( mech, approach_node, tcf );

SEE ALSO

CxTeachNode
CxSafeJumpMove

Performs a safe move from the current location to a target location

SYNOPSIS

#include <code.robpac.h>
long CxSafeJumpMove(CxMechanism mech, long z_axis_joint_number,
                     CxNodeId target_base, CxNodeId tcf, double x, double y, double z ,
                     double theta, CxNodeId safe, double trigger_offset, double
                     complete_offset )

ARGUMENTS

mech

Mechanism ID. This value is returned by a call to CxOpenMechanism.

z_axis_joint_number

User-specified number of the joint which corresponds to the z-axis. This
joint should be translational and should correspond to only one motor axis.

target_base

ID of the node which establishes to the base target position. This node
does not necessarily constitute the target position itself though, as
explained in the discussions of arguments x, y, z and theta.

tcf

Node ID of the tool control frame

x, y, z, theta

Offsets from the target node which mark the actual target position. The
translational offsets x, y and z are measured along the cartesian axes of the
target node, while theta is measured around the target node's z-axis.

safe

ID of a node which lies in the so-called "safe plane," above which all
horizontal robot motion is considered to be safe.

trigger_offset

See Definitions section below.

complete_offset

See Definitions section below.

DESCRIPTION

The safe jump move intends to provide an extremely quick and efficient method to perform a motion profile
common to most pick and place applications and similar applications. The target speed of the entire move is
determined by the current % joint speed. The acceleration is determined by the current acceleration and
deceleration times.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
CxSetAccelType

Sets the mechanism ramp acceleration type

SYNOPSIS

#include <code/robpac.h>
long CxSetAccelType(CxMechanism mech, long type)

ARGUMENTS
mech    Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
type    Mechanism acceleration type. See DESCRIPTION below for possible types.

DESCRIPTION

This function sets the mechanism ramp acceleration type for a given mechanism. The two supported acceleration types are defined as follows:

- CX_CONST_RAMP_TIME: The time set for acceleration is a constant, thus the acceleration magnitude will be determined by the speed change required.
- CX_CONST_RAMP_ACCEL: A constant acceleration is used to accomplish speed change. Thus the time will vary depending on the speed change.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ACCEL_TYPE</td>
<td>Value entered is not CX_CONST_RAMP_ACCEL or CX_CONST_RAMP_TIME.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

SEE ALSO

CxGetAccelType
CxSetBlendPolicy

Sets the mechanism’s current blend policy to one of five modes

SYNOPSIS

```c
#include <code/robpac.h>
long CxSetBlendPolicy(CxMechanism mech, long policy)
```

ARGUMENTS

- `mech`: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `policy`: Policy which is used to determine whether blending is to be used. See DESCRIPTION below for possible blend policies.

DESCRIPTION

This function is used to set the specified mechanism’s current blend policy. The blend policy defines whether or not a motion comes to a complete stop before initiating the next move in the Server’s motion queue. It also defines if a motion related function call will return immediately after the move is placed in the motion queue or wait until the move has finished processing. Note that CIMControl does not support blended moves with different TCF’s.

Possible values for the blend policy are defined as follows:

- **CX_MOVE_TO**
  This policy will load a move into a move queue and return immediately (it will not wait for the motion to complete). A motion command with this blend policy will come to a complete stop before initiating the next move in the queue.

- **CX_MOVE_THRU**
  This policy will load a move into a move queue and return immediately (it will not wait for the move to complete). Motion commands with this blend policy will not come to a complete stop if there is another move following in the queue. Instead, the next move in the queue will be blended with the current move.

- **CX_MOVE_WAIT**
  Motion commands with this blend policy will load a move into the queue and will wait until all previously queued moves and the current commanded move are completed or aborted before returning. Motions with this blend policy will also come to a complete stop before returning. The default is **CX_MOVE_WAIT**.

- **CX_MOVE_TO_TANGENT**
  This blend policy should be applied to paths of the type `curveseg` (linked list of curvilinear segments). Tangent segments will be transitioned exactly without blending, maintaining the desired speed, while non-tangent segments will have their blend policy set to **CX_MOVE_TO**.

- **CX_MOVE_THRU_TANGENT**
  This blend policy should be applied to paths of the type `curveseg` (linked list of curvilinear segments). Tangent segments will be transitioned exactly without blending, maintaining the desired speed, while non-tangent segments will have their blend policy set to **CX_MOVE_THRU**.

See the `CODE Applications Programming` manual for a description of blended and non-blended moves.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is not CX_MOVE_TO, CX_MOVE_WAIT, or CX_MOVE_THRU.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first sets the blend policy to CX_MOVE_WAIT and then issues commands to move joints. Then it gets the blend policy for the mechanism and prints it out. The print message appears only after both the moves are complete. Then it sets the blend policy to CX_MOVE_TO and does the same thing. This time the message appears even though the motion is not complete.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;
    long policy;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_WAIT);
    CxMoveSingleAxis( mech, 0, 100.0 );
    CxMoveSingleAxis( mech, 1, 120.0 );
    CxGetBlendPolicy( mech, &policy );
    printf(" Current policy is : %ld \n", policy );

    CxSetBlendPolicy( mech, CX_MOVE_TO);
    CxMoveSingleAxis( mech, 0, 0.0 );
    CxMoveSingleAxis( mech, 1, 0.0 );
    CxGetBlendPolicy( mech, &policy );
    printf(" Current policy is : %ld \n", policy );
    CxWaitForEndOfMotion( mech );

    CxRobpacExit();
}
```

**WARNINGS**

The user must be sure that the motion card is capable of blending before using CX_MOVE_THRU. Currently, this mode is only supported for use with the PMAC and MEI-DSP motion cards.
HARDWARE AND SYSTEM DEPENDENCIES

Move types and blending capabilities depend on the controller hardware and software. It is not safe to assume that all controllers support these move capabilities. With CODE 4.0, CIMControl supports blended moves on the PMAC and MEI-DSP motion cards, but no on the MEI-XMP motion card.

SEE ALSO

CxGetBlendPolicy
**CxSetBlendType**

Sets the mechanism’s current blend type

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetBlendType(CxMechanism mech, long type)
```

**ARGUMENTS**

- `mech`  
  Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

- `type`  
  Desired blend type. See **DESCRIPTION** below for possible blend types.

**DESCRIPTION**

This function is used to set the specified mechanism’s blend type. The blend type is specified as one of two types:

- `CX_DECEL_BLEND`  
  This is the normal default blend type, which starts blending two moves when the first move begins its deceleration period to zero speed. The second move begins its accel period at the exact time the first begins its decel period and both moves are integrated together to maintain the speed near the desired speed setting.

- `CX_PERCENT_SPEED_BLEND`  
  This blend type uses the decel blend type except the accel period of the second blended move is delayed until the speed of the first move is reduced to some percentage of the speed set for the move. Another API function, CxSetPercentBlendSpeed(), is required to set the percent of the set speed at which blending is to occur; otherwise, the default percentage of 100% is used.

Note that the user enters a number between 0.0 and 1.0 rather than an actual percentage. The default value is 1.0. See **CODE Applications Programming** for a description of blended and non-blended moves.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is not CX_DECEL_BLEND or CX_PERCENT_SPEED_BLEND</td>
</tr>
<tr>
<td>CX_Mech_Not_Open_For_Control</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following program sets the tool speed to 100 mm/s, then sets the blend type to CX_PERCENT_SPEED_BLEND, and then uses the API function CxSetPercentBlendSpeed() to set the blend speed at 50% of the desired speed setting. This means that the blending of two moves will begin when the speed of the first move falls below 50 mm/s.

```
#include <stdio.h>
#include <code/robpac.h>

void main ( void ) {
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100;
    long policy;
    
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
    CxSetToolMotionType ( mech, CX_LINEAR_INTERP);
    CxSetTipSpeed ( mech, 100.0 );
    CxSetBlendPolicy ( mech, CX_MOVE_THRU );
    CxSetBlendType ( mech, CX_PERCENT_SPEED_BLEND );
    CxSetPercentBlendSpeed ( mech, 0.5 );
    
    CxWaitForEndOfMotion ( mech ) ;
    robpac_exit();
}
```

WARNINGS

The user must be sure that the motion card is capable of blending in order to take advantage of this function. Currently, CIMControl only supports blending on the PMAC and MEI-DSP motion cards.

HARDWARE AND SYSTEM DEPENDENCIES

Move types and blending capabilities depend on the controller hardware and software. It is not safe to assume that all controllers support these motion capabilities. With CODE 4.0, CIMControl supports blended moves on the PMAC and MEI-DSP motion cards, but not on the MEI-XMP motion card.

SEE ALSO

CxGetBlendPolicy, CxSetPercentBlendSpeed, CxGetBlendType
**CxSetInterpType**

Sets the motion interpolation type to be used in target moves

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetInterpType(CxMechanism mech, long interp_type)
```

**ARGUMENTS**

- `mech` Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `interp_type` Current motion interpolation type. See **DESCRIPTION** below for possible interpolation types.

**DESCRIPTION**

This function sets the mechanism’s current interpolation type for subsequent moves. The interpolation type determines the path that the mechanism will follow when moving from one configuration to another. The function can be one of the following:

- `CX_JOINT_INTERP` The most efficient setting. Motors do not reverse direction or change speeds during a move. All joints arrive at new settings simultaneously.
- `CX_LINEAR_INTERP` Moves TCF in a straight line from current pose to specified pose.
- `CX_LINEAR_JOG` Moves TCF in straight line in direction specified until an asynchronous event occurs (i.e., error, `CxStopMechanism`, etc.).

A path move or a circular move supersedes this setting.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Invalid motion interpolation type.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the interpolation type to `CX_JOINT_INTERP`, then sets the joint speed to 40% of the maximum. Next, it starts motion and goes in a loop. Finally, it continuously prints out the actual interpolation type and actual speed.

```c
#include <stdio.h>
#include <code/robpac.h>
```
void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId MyMech, tcf;
    CxNodeId t1, t2, t3;
    double actual_speed;
    long interp_type;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO );
    CxSetInterpType( mech, CX_JOINT_INTERP );
    CxSetJointSpeed( mech, 0.4 );

    /* Move MyMech to perform a task */
    CxMoveToNode( mech, t1, tcf );
    CxMoveToNode( mech, t2, tcf );
    CxMoveToNode( mech, t3, tcf );

    /* Continuously observe MyMech parameters */
    while (1)
    {
        CxGetActualInterpType( mech, MyMech, &interp_type );
        CxGetActualJointSpeed( mech, MyMech, &actual_speed );
        printf(" Current interpolation type is : %ld \n", interp_type );
        printf(" Current tcf node speed is     : %lf \n", actual_speed );
    }
    
    CxRobpacExit();
}

WARNINGS
This function affects the path that a mechanism will take when traveling between poses.

HARDWARE AND SYSTEM DEPENDENCIES
Not all mechanisms support inverse kinematics, which means that they may only be capable of joint-interpolated moves.

SEE ALSO
CxGetInterpType, CxGetActualInterpType
**CxSetInvkinSoln**
Sets the desired inverse kinematic solution number

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetInvkinSoln(CxMechanism mech, long soln_num)
```

**ARGUMENTS**

- **mech**: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **soln_num**: Mechanism inverse kinematic solution number, default is -1 (`CX_ALL_SOLN`).

This function is used to set the mechanism inverse kinematic solution number for a given mechanism. If the solution number is set to `CX_ALL_SOLN` (-1), the mechanism will calculate all possible solutions and choose one that requires the smallest joint changes. Undesirable arm flip may occur when `CX_ALL_SOLN` is used. Mechanism controllers usually use only one solution to get rid of the flip situation. If `soln_num` falls outside the valid solution numbers for the mechanism, the future motion related function calls may fail.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following example sets the invkin solution to 0 before executing some inverse kinematic moves:

```c
CxServer Server;
CxMechanism s100;
long soln_num;
CxNodeId t1_id, t1_id, s100_id, t2_id, t3_id;

/* open server */
Server = CxOpenServer("Server", CX_SMEM, 0);

/* get node id’s */
```
CxGetNamedNodeID(Server,"t1", &t1_id);
CxGetNamedNodeID(Server,"t2", &t2_id);
CxGetNamedNodeID(Server,"s100", &s100_id);
CxGetNamedNodeID(Server,"t3", &t3_id);
CxGetNamedNodeID(Server,"t4", &t4_id);

/* open mechanism */
s100 = CxOpenMechanism(Server, s100_id, CX_CONTROL);

CxSetInterpType(s100, CX_JOINT_INTERP);

/* set solution number and move to different targets */
CxSetInvkinSoln(s100, 0);
CxMoveToNode(s100, t1, tcf);
CxMoveToNode(s100, t2, tcf);
CxMoveToNode(s100, t3, tcf);
CxMoveToNode(s100, t4, tcf);

SEE ALSO
CxGetInvkinSoln, CxGetActualInvkinSoln
CxSetJacobian

Sets the Jacobian speed checking flag on or off

SYNOPSIS

```
#include <code/robpac.h>
long CxSetJacobian(CxMechanism mech, long flag)
```

ARGUMENTS

- **mech**: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **flag**: A flag which toggles Jacobian checking (acceptable values: CX_ON or CX_OFF). Default setting is CX_ON.

DESCRIPTION

The function CxSetJacobian turns Jacobian checking on or off for singularity or redundancy checking. If off, the TCF speed in path following motion will remain constant and not be reduced to account for directional changes. It is useful to turn the Jacobian CX_OFF for mechanisms which have redundant joints and for which you do not have a good model of the way the controller handles the redundancies during path following moves. The default setting is CX_ON.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_PARM</td>
<td>The flag entered is not CX_ON or CX_OFF.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

WARNINGS

Jacobian checking must be turned on to determine the speeds and servo time increments for path following moves.

HARDWARE AND SYSTEM DEPENDENCIES

It is often difficult to determine how controller vendors treat the joint control of joints that are colinear (redundant). In the absence of an accurate model, it may be useful to turn Jacobian checking CX_OFF and assume that the cycle times are only approximate. Otherwise, the CODE Jacobian and speed utilities may interrupt or halt the motion sequences.

SEE ALSO

CxGetJacobian
CxSetJointMoveTime
Determine the time to make a joint move

SYNOPSIS
```
#include <code/robpac.h>
long CxSetJointMoveTime(CxMechanism mech, long err_mode, double move_time)
```

ARGUMENTS
- **mech**: Mechanism IF of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **err_mode**: If CX_TRUE, server will return an error and stop motion if can't achieve move time. If CX_FALSE, server will set acceleration parameters to maximum and achieve best move time possible.
- **move_time**: Time desired for the next joint move.

DESCRIPTION
This function sets the move time desired for the next joint move (and only the next joint move) which follows this function. The next move must be a joint move and must immediately follow this function. If the error mode is CX_TRUE and the motion parameters (acceleration values, joint speed settings, etc.) are not sufficient for the mechanism to achieve the final joint values in the desired move time, the server will return an error. If the error mode is CX_FALSE, the mechanism will use the maximum motion parameters to achieve the closest possible move time to that specified.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_CANT_ACHIEVE_MOVE_TIME</td>
<td>If err_mode is set to CX_TRUE and move time can’t be achieved, then this error will be returned.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following code will set the move time as 3.2 seconds for a joint move from the current configuration to the joint values specified in dof:

```c
CxMechanism mech;
double dof[CX_MAXJOINTS];

CxSetJointMoveTime(mech, CX_FALSE, 3.2);
CxMoveAllAxes(mech, dof);
```
SEE ALSO

CxGetJointMoveTime
**CxSetJointSpeed**

Sets the mechanism’s joint speed to a percentage of its maximum

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetJointSpeed(CxMechanism mech, double speed)
```

**ARGUMENTS**

- `mech` Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `speed` Speed at which the joint is to move. Speed must be between 0.0 and 1.0, where 0.5 would represent 50% of the mechanism’s maximum speed.

**DESCRIPTION**

The function `CxSetJointSpeed` sets the percentage of a mechanism’s maximum joint speed that will be used in subsequent joint moves. The joint speed is used only when the motion interpolation type is set to `CX_JOINT_INTERP`.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (`CX_ERROR`) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE.Receive_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is less than zero or negative or greater than 1.0.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the interpolation type to `CX_JOINT_INTERP`, then sets the joint speed to 40% of the maximum. It then starts motion and goes into a loop. Finally, it continuously prints out the actual interpolation type and actual speed.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId MyMech, tcf;
```
CxNodeId t1, t2, t3;
double actual_speed;
long interp_type;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );

mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO );

CxSetInterpType( mech, CX_JOINT_INTERP );
CxSetJointSpeed( mech, 0.4 );

/* Move MyMech to perform a task */
CxMoveToNode( mech, t1, tcf );
CxMoveToNode( mech, t2, tcf );
CxMoveToNode( mech, t3, tcf );

/* Continuously observe MyMech parameters */
while (1)
{
    CxGetActualInterpType( mech, MyMech, &interp_type );
    CxGetActualJointSpeed( mech, MyMech, &actual_speed );
    printf(" Current interpolation type is : %ld \n" , interp_type );
    printf(" Current tcf node speed is : %lf \n" , actual_speed);
}

CxRobpacExit();

WARNINGS

When executing a program on-line for the first time, it’s a good idea to set the joint speed to a slower value than when executing a finished, well-tested program.

HARDWARE AND SYSTEM DEPENDENCIES

While some mechanism vendors allow joint speed ranges, which are greater than 100%, CODE maps the speed range between 0.0 and 1.0. For example, if a vendor-specified joint speed ranges from 0 to 200, a joint speed setting of 1.0 would indicate 200 and a speed of 0.5 would indicate 100.

SEE ALSO

CxSetInterpType, CxGetJointSpeed, CxGetActualJointSpeed
**CxSetMotionTrack**

Sets the motion tracking flag on or off

### SYNOPSIS

```c
#include <code/robpac.h>
long CxSetMotionTrack(CxMechanism mech, long flag)
```

### ARGUMENTS

- **mech**
  Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

- **flag**
  Set to CX_ON or CX_OFF.

### DESCRIPTION

The function turns motion tracking **CX_ON** or **CX_OFF**. When motion tracking is **CX_ON**, the robot will continually update the target value from the database and converge to the moving target (if robot speed higher than target speed). In effect, the CODE initialization routines are called each servo cycle and will slow down the trajectory calculations. The robot will track every target except those physically attached to the robot for which tracking is turned on. The default setting is **CX_OFF**.

### RETURN VALUES

This function returns **0** if successful; otherwise, **-1 (CX_ERROR)** is returned.

### ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

### EXAMPLE

The following program chases a moving object. It sets the motion tracking **CX_ON**. Then it moves the TCF relative to the moving object. As soon as TCF reaches the specified point, the program picks up the moving target and sets tracking **CX_OFF**. Then it gets back the tracking setting and checks it to make sure that it is set to **CX_OFF**.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId s100, moving_target, tcf;
```
Long is_still_tracking;

Server = CxOpenServer(Server, s100, CX_CONTROL);
CxSetBlendPolicy(mech, CX_MOVE_WAIT);
.
./
/* set the target in motion */
/* Chase moving object and pick it up*/
CxSetMotionTrack(mech, CX_ON);
CxMoveRelNode(mech, moving_target, tcf, "XYZ", 0.0, 0.0, 0.0, 5.0, 5.0, 5.0);
Pickup_part(); /* this is not a code function */
CxSetMotionTrack(mech, CX_OFF);
CxGetMotionTrack(mech, &is_still_tracking);
If (is_still_tracking == CX_ON)
{
    CxStopMechanism(mech);
    Printf("Problem with tracking. Abort in move... 
");
}
.
.
CxRobpacExit();
}

WARNINGS

Motion tracking will slow down the trajectory calculations and should only be used when you wish to track a particular moving target; thus, it is wise to turn motion tracking CX_ON only when you move to a moving target. After target acquisition and task completion, you should then turn the tracking CX_OFF.

HARDWARE AND SYSTEM DEPENDENCIES

Motion tracking will usually require special hardware such as vision to track a moving target and update the CODE database. In addition, it is usually wise to write a “CODE API” process that uses a predictor/corrector algorithm to predict where the target will move and then turn motion tracking on at the appropriate time for interception.

SEE ALSO

CxGetMotionTrack
**CxSetMoveBufferSize**

Sets the move buffer size on the servocard being used.

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetMoveBufferSize(CxMechanism mech, long size)
```

**ARGUMENTS**

- `mech`  
  Mechanism ID; This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `size`  
  Sets desired buffer size on servocard.

**DESCRIPTION**

This function sets the current buffer size being used on the motion card. Buffers are used to queue a sequence of moves so that the card can blend them together or improve cycle time by reducing the time latency between adjacent moves for cycle time improvement. Usually, this buffer size will be set to some number between 3 and 10. The buffer size that is entered must be less than or equal to the CX_MAX_MOVE_BUFFER_SIZE (currently set at 4). The default buffer size for non-VecTool motion processing is 4 for PMAC and 7 for MEI.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program determines the current size of the move buffer. If it is greater than 2, then it sets the buffer size to 2.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId MyMech;
    long size;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism ( Server, MyMech, CX_CONTROL );
```
CxGetMoveBufferSize( mech, &size);

if(size > 2)
{
    CxSetMoveBufferSize( mech, 2);
}

SEE ALSO

CxGetMoveBufferSize
**CxSetMoveTangent**

Sets the value for which two moves are considered tangent

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetMoveTangent (CxMechanism mech, double tang_value)
```

**ARGUMENTS**

- `mech` Mechanism ID:
  - This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `size` Sets desired buffer size on servocard.

**DESCRIPTION**

`CxSetMoveTangent` sets the value at which two moves are considered tangent. This function is used for the special blend policies of `CX_MOVE_TO_TANGENT` or `CX_MOVE_THRU_TANGENT` which seek to maintain the exact path and speed setting for paths that are tangent to each other. In contrast to the blend policy of `CX_MOVE_THRU`, motion between tangent moves will not blend the decel period of the first move with the accel period of the second move. Normally, the user would not want to set the tangent value to more than 5 degrees, otherwise, significant path deviation will occur. Currently, this function can only be used in evaluating tangency between `curveseg` moves and NC feed moves.

A move in which the tool orientation dominates curvilinear displacement (position change) will be considered non-tangent to the previous move.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the tangent value used in checking `curveseg` tangencies to 1 degree, then sets the blend policy to `CX_MOVE_THRU_TANGENT` for a move along a `curveseg`.

```c
#include <stdio.h>
#include <code/robpac.h>

void pickup_part();

void main ( void )
{
```
CxServer Server;
CxMechanism mech;
CxNodeId   s100, path, tcf;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
.
mech = CxOpenMechanism( Server, s100, CX_CONTROL );

CxSetMoveTangent ( mech, 1.0 ) ;
CxSetBlendPolicy ( mech, CX_MOVE_THRU_TANGENT ) ;
.
CxMoveRelPath ( mech, path, tcf,"","", "XYZ", 0.0, 0.0, 0.0,
   0., 0.,0. );
.
CxRobpacExit();
}

SEE ALSO
    CxGetMoveTangent
CxSetPathSegAccel
Sets the acceleration value along one segment of a path

SYNOPSIS
#include <code/robpac.h>
long CxSetPathAccel (CxServer server, CxNodeId node, char *seg_name, double accel);

ARGUMENTS
server Server ID
node Curve node ID
seg_name Name of the segment
accel Value used for acceleration (deceleration) along the segment

DESCRIPTION
This function sets the acceleration (deceleration) value used in generating the velocity profile for motion along a segment of a path.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
**CxSetPathSegmentSignal**

Sets the start and stop signals for one segment of a path

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetPathSegmentSignal (CxServer server, CxNodeId node, char *seg_name, long start_signal, long start_sig_value, long end_signal, long end_sig_value);
```

**ARGUMENTS**

- `server` Server ID
- `node` Curve node ID
- `seg_name` Name of the segment
- `start_signal` Start signal ID
- `start_sig_value` Start signal value
- `end_signal` End signal ID
- `end_sig_value` End signal value

**DESCRIPTION**

This function enables the motion start signal and motion end signal when moving along a given path.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
**CxSetPathSegVelocity**

Sets the tip velocity along one segment of a path

**SYNOPSIS**

```
#include <code/robpac.h>

long CxSetPathVelocity (CxServer server, CxNodeId node, char *seg_name, double velocity);
```

**ARGUMENTS**

- **server**: Server ID
- **node**: Curve node ID
- **seg_name**: Name of the segment
- **velocity**: Value used for velocity along the segment

**DESCRIPTION**

This function sets the tip velocity used in generating the velocity profile for motion along a segment of a path. If the previous segment has a different velocity, the tip will accelerate (decelerate) based on the current acceleration value and machine parameters.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
**CxSetPercentBlendSpeed**
Sets the percent of set speed at which blending begins

**SYNOPSIS**
```c
#include <robline/robpac.h>
long CxSetPercentBlendSpeed (CxMechanism mech, double fraction)
```

**ARGUMENTS**
- `mech`  Mechanism ID;
  This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `fraction`  Sets the desired buffer size on servocard.

**DESCRIPTION**
This function will set the fractional percent of full speed at which blending of two moves will occur. If, for example, the fractional percent setting is 0.8, then a second move will begin to blend with the first move when the latter has decelerated to 80% of the maximum speed set for the move.

This function will not return an error if the number entered is negative or greater than 1.0, rather, the fractional percent value will be set to the default value of 1.0 instead. See the Applications Programming manual for a description of blended and non-blended moves.

**RETURN VALUES**
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
The following program sets the tool speed to 100 mm/s, then sets the blend type to CX_PERCENT_SPEED_BLEND, and then uses the API function CxSetPercentBlendSpeed() to set the blend speed at 50% of the desired speed setting. This means that the blending of two moves will begin when the speed of the first move falls below 50 mm/s.

```c
#include <stdio.h>
#include <code/robpac.h>

void main ( void )
{
    CxServer    Server;
    CxMechanism mech;
```
CxNodeId s100;
long policy;

Server = CxOpenServer ( "Testing", CX_SMEM, 0 );
  .
mech = CxOpenMechanism ( Server, s100, CX_CONTROL );
CxSetToolMotionType ( mech, CX_LINEAR_INTERP );
CxSetTipSpeed ( mech, 100.0 );
CxSetBlendPolicy ( mech, CX_MOVE_THRU );
CxSetBlendType ( mech, CX_PERCENT_SPEED_BLEND );
CxSetPercentBlendSpeed ( mech, 0.5 );
  .
  .
CxWaitForEndOfMotion ( mech );

CxRobpacExit ( ) ;
}

WARNINGS
The user must be sure that the motion card is capable of blending in order to take advantage of this function. Currently, CIMControl only supports blending on the PMAC and MEI-DSP motion cards.

HARDWARE AND SYSTEM DEPENDENCIES
Move types and blending capabilities depend on the controller hardware and software. It is not safe to assume that all controllers support these move capabilities. With CODE 4.0, CIMControl supports blended moves on the PMAC and MEI-DSP motion cards, but not on the MEI-XMP motion card.

SEE ALSO
CxGetBlendPolicy, CxGetPercentBlendSpeed, CxSetBlendType
**CxSetSAccelTimes**

Sets the desired S ramp acceleration and deceleration times

**SYNOPSIS**

```
#include <code/robpac.h>
long CxSetSAccelTimes(CxMechanism mech, double rise_S_time, double fall_S_time)
```

**ARGUMENTS**

- `mech` : Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `rise_S_time` : Amount of time for constant jerk portion of the acceleration phase.
- `fall_S_time` : Amount of time for constant jerk portion of the deceleration phase.

**DESCRIPTION**

This function is used to set the current S-curve acceleration and deceleration times. These times are used if the mechanism’s current acceleration type is **CX_CONST_RAMP_TIME** (see CxGetAccelType, CxSetAccelType). An S-curve acceleration profile is useful for minimizing the wear on a mechanism because the forces being applied to the mechanism’s gear train are changed gradually through the acceleration period.

When a **CX_CONSTANT_RAMP_TIME** acceleration setting is used, the S-curve acceleration and deceleration profiles are combined with the constant trapezoidal acceleration and deceleration times respectively (see CxGetTrapAccelTimes, CxSetTrapAccelTimes).

The following illustrations show the relationship of the trapezoidal acceleration times and the S curve acceleration times. The figures include motion with no S-curve acceleration (pure trapezoidal acceleration profile), only S-curve acceleration, and combined S-curve and trapezoidal acceleration profiles. Note that in these figures, the acceleration profiles are the same as the deceleration profiles (i.e. the acceleration times are the same as the deceleration times); therefore, `S_time` is used to represent both. Likewise, `rise_time` and `fall_time` are used to represent trapezoidal acceleration times.

In order for the S-curve acceleration parameter to be used when operating in **CX_RUNTIME** mode, the underlying motion control interface must support S-curve acceleration profiles (see HARDWARE AND SYSTEM DEPENDENCIES below).
NOTE: If the specified value for rise_S_time is greater than ½ the rise_time, the rise_S_time will be set to ½ the rise_time.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>A time entered is zero or negative or the time entered is less than 0.001 sec.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

WARNINGS

The user must be extremely cautious when entering these values if they will be used to actually control a mechanism since allowable accelerations are mechanism and design dependent. Excessive accelerations may cause the mechanism’s inertial loads to damage the drive transmissions and other mechanical components on the mechanism.

EXAMPLE

The following program opens a mechanism for CX_CONTROL. It first sets the trajectory accel type to CX_TRAP_ACCEL for the first move, then changes it to CX_S_ACCEL for the second move.

```c
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    long      intrp_type, traj_accel_type;
    CxNodeId   robot, tcf, t1, t2;

    Server = CxOpenServer( "Testing", CX_SYSTEM_V, 0 );
    CxGetNamedNodeId( Server, "robot", &robot );
    CxGetNamedNodeId( Server, "tcf", &tcf );
```
CxGetNamedNodeId( Server, "t1", &t1 );
CxGetNamedNodeId( Server, "t2", &t2 );

mech = CxOpenMechanism( Server, robot, CX_CONTROL );

CxSetTrajAccelType( mech, CX_TRAP_ACCEL );
CxMoveToNode( mech, t1, tcf );
CxSetTrajAccelType( mech, CX_S_ACCEL );
CxMoveToNode( mech, t2, tcf );

CxWaitForEndOfMotion( mech );
CxRobpacExit();
**CxSetScrewSpeed**

Sets the desired screw speed in curvilinear moves

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetScrewSpeed(CxMechanism mech, double speed)
```

**ARGUMENTS**

- **mech**  
  Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

- **speed**  
  Desired screw speed, in deg/sec or rad/sec depending on the units specified via CxSetUnit.

**DESCRIPTION**

This function sets the current screw speed setting for the given mechanism. The higher the speed, the faster the tool will rotate.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is zero or negative or less than 0.001.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

CxGetActualScrewSpeed, CxGetScrewSpeed
CxSetStartSignal

Sets the motion start signal of the given mechanism

SYNOPSIS
#include <code/robpac.h>
long CxSetStartSignal(CxMechanism mech, long start_signal, long sig_value)

ARGUMENTS
mech Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

start_signal Signal ID
sig_value Signal value

DESCRIPTION
This function enables the motion start signal of the given mechanism. A motion start signal can be used to detect the beginning of a non-blended move for the given mechanism.

If a start signal is set using the CxSetStartSignal, the given signal will be set to the specified value whenever motion begins. In order to ensure that the start signal is set when motion starts, the previous move command should have either a CX_MOVE_TO or CX_MOVE_WAIT blend policy. CxClearStartSignal disables the start signal. Note that the start signal is not set during blended motions.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program first initializes the software signal SW_SIG1 to 0. Then it sets up an interrupt handler for that signal in such a way that whenever the signal value reaches 1 that interrupt handler will be called. Then it designates signal SW_SIG1 as a start signal. Now whenever motion starts, that signal will be set to “1”. Then it moves an axis to a joint value of 100.0. As motion starts the interrupt handler gets invoked, then the program waits until the motion is complete. It then clears the start signal. So from the next move on signal SW_SIG1 does not get set to value 1 and the interrupt handler does not get called.

#include <stdio.h>
#include <code/robpac.h>

#include "sigTable" /*signal table constant definitions */

void motion_started (long signal, long value, CxMechanism mech);

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_WAIT);

    CxSetSignal(Server, SW_SIG1, 0);
    interrupt_on_value( Server, SW_SIG1, 1, mech, motion_started);
    CxSetStartSignal(mech, SW_SIG1, 1);

    CxMoveSingleAxis( mech, 0, 100.0 );

    CxClearStartSignal( mech );
    CxMoveSingleAxis( mech, 0, 0.0 );

    CxRobpacExit();
}

void motion_started(long signal, long value, CxMechanism mech)
{
    CxSendMechanismErrorAction( mech, CX_MECH_RESUME);
    printf(" Motion Started \n");
    CxSetSignal( Server, SW_SIG1, 0);
}

SEE ALSO

CxClearStartSignal, CxSetStopSignal
**CxSetStopSignal**

Sets the motion stop signal of the given mechanism

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetStopSignal(CxMechanism mech, long stop_signal, long sig_value)
```

**ARGUMENTS**

- **mech**: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **stop_signal**: Signal ID
- **sig_value**: Signal value

**DESCRIPTION**

This function enables the motion stop signal of the given mechanism. A stop signal can be used to detect when a motion has successfully reached a target position.

If a stop signal is set, the given signal will be set to the specified value whenever a motion command reaches the target location. This action will occur until the `CxClearStopSignal` function is called. The stop signal will not be triggered if the current move has a `CX_MOVE_THRU` blend policy or if the motion is aborted (see `CxSendMechanismErrorAction`). In order to ensure that the stop signal is set when a target is reached, the move command should have either a `CX_MOVE_TO` or `CX_MOVE_WAIT` blend policy.

Note that the stop signal is not set during blended motions.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first initializes the software signal SW_SIG2 to 0.0. Next, it sets up an interrupt handler for that signal in such a way that whenever the signal value reaches 1 that interrupt handler will be called. It designates signal SW_SIG2 as a stop signal so that whenever motion ends, that signal will be set to value 1. Then it moves an axis to a joint value of 100.0. As soon as motion ends for one move, the interrupt handler gets invoked. After the first move, the program clears the stop_signal, and from the next move on, signal SW_SIG2 does not get set to value 1 and the interrupt handler does not get called after the motion is completed.

```c
#include <stdio.h>
```
#include <code/robpac.h>

#include "sigTable" /*signal table constant definitions */
void motion_ended(long signal, long value, CxMechanism mech);

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_WAIT);

    CxSetSignal(Server, SW_SIG2, 0);
    interrupt_on_value( Server, SW_SIG2, 1, mech, motion_ended);
    CxSetStopSignal(mech, SW_SIG2, 1);

    CxMoveSingleAxis( mech, 0, 100.0 );
    CxClearStopSignal( mech );
    CxMoveSingleAxis( mech, 0, 0.0 );
    CxRobpacExit();
}

void motion_ended(long signal, long value, CxMechanism mech)
{
    CxSendMechanismErrorAction( mech, CX_MECH_RESUME);
    printf(" Motion Ended 
");
    CxSetSignal( Server, SW_SIG2, 0);
}

SEE ALSO

CxClearStopSignal,CxSetStartSignal
CxSetTipSpeed

Sets the desired tool/tip linear speed for curvilinear moves

SYNOPSIS

```c
#include <code/robpac.h>
long CxSetTipSpeed(CxMechanism mech, double speed)
```

ARGUMENTS

- **mech**: Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **speed**: Linear speed in linear units/sec.

DESCRIPTION

This function sets the linear speed for subsequent curvilinear moves. This value is ignored for moves with the motion interpolation type set to `CX_JOINT_INTERP`.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is less than 0.001.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

WARNINGS

When executing a program in runtime mode for the first time, it's a good idea to set this speed to a slower value than when executing a finished well-tested program.

The function `CxSetTipSpeed` has no effect on joint-interpolated moves.

HARDWARE AND SYSTEM DEPENDENCIES

This parameter has no effect on mechanisms, which do not support inverse kinematics. Units are mechanism dependent.

SEE ALSO

`CxGetTipSpeed`, `CxSetInterpType`, `CxGetActualToolSpeed`
**CxSetToolMotionType**
Sets the tool motion type

**SYNOPSIS**
```
#include <code/robpac.h>
long CxSetToolMotionType(CxMechanism mech, long tool_motion_type)
```

**ARGUMENTS**
- mech: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- Tool_motion_type: Tool motion type to be set. See DESCRIPTION below for definitions of possible tool motion types.

**DESCRIPTION**
This function sets tool motion type. By default, the various move commands cause the tool to orient itself exactly as specified during a move. However, not all tools require strict orientation. Currently four tool motion types can be set for a tool attached to a mechanism.

- **CX_FULL_POSE**: Tool must align exactly as specified (the default).
- **CX_FIXED_ORIENT**: Tool must maintain its current orientation.
- **CX_Z_POS**: Mechanism must be capable of interpolating the tool TCF Z axis in a plane parallel to the tool’s and target’s Z axes.
- **CX_Z_POSE_NO_SPIN**: Tool Z interpolation occurs in plane parallel to tool and target Z axes but no interpolation spin about Z axis to align tool X,Y axes with target X,Y axes.

In CIMServer a **CX_SERIAL_ROBOT** is treated differently from an **CX_NC_ROBOT** when using the **CX_Z_POSE** type. The **CX_SERIAL_ROBOT** assumes that the tool can be spun around the tool Z axis to align the tool X,Y axes with the target X,Y axes. The **CX_NC_ROBOT** cannot; thus, if the mechanism is an **CX_NC_ROBOT** and **CX_Z_POSE** is specified, then the type defaults to **CX_Z_POSE_NO_SPIN**.

**RETURN VALUES**
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the CxGetErrorCode function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is not CX_FULL_POSE, CX_FIXED_ORIENT, or CX_Z_POSE.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a mechanism for `CX_CONTROL`, then checks the interpolation type. If this is set to `CX_LINEAR_INTERP`, the program checks the tool motion type. If the tool motion type is set to `CX_FULL_POSE`, the program resets it to `CX_FIXED_ORIENT`.

```c
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    long       intrp_type, tool_motion_type;
    CxNodeId   MyMech, tcf, t1;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );

    mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO );

    CxGetInterpType( mech, &intrp_type );
    if (intrp_type == CX_LINEAR_INTERP)
    {
        CxGetToolMotionType( mech, &tool_motion_type );
        if (tool_motion_type == CX_FULL_POSE)
            CxSetToolMotionType( mech, CX_FIXED_ORIENT );
    }
    CxMoveToNode( mech, t1, tcf );

    CxWaitForEndOfMotion( mech );
    CxRobpacExit();
}

SEE ALSO

CxSetToolMotionType
**CxSetTrajAccelType**

Sets the type of trajectory acceleration or deceleration profile

**SYNOPSIS**

```
#include <code/robpac.h>
long CxSetTrajAccelType(CxMechanism mech, long type)
```

**ARGUMENTS**

- **mech** Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **type** Trajectory profile type to be set. See DESCRIPTION below for definitions of possible types.

**DESCRIPTION**

This function sets trajectory acceleration or deceleration profile type. The profile describes the shape of the velocity versus time curve as the mechanism changes either Cartesian or joint speed. The default type is `CX_TRAP_ACCEL` in which the speed changes linearly with time. The `CX_S_ACCEL` type uses constant jerk S sections to transition the speed change smoothly. Currently two profile types can be set for mechanism motion.

- `CX_TRAP_ACCEL` Trapezoidal profile.
- `CX_S_ACCEL` Uses constant jerk S transitions to change speed.

If the accel type is specified as `CX_CONST_RAMP_ACCEL` and the profile type is `CX_S_ACCEL`, then the time to perform the speed change is calculated using the ramp acceleration set for this move. If the accel type is `CX_CONST_RAMP_TIME` and the profile type is `CX_S_ACCEL`, then the S accel times set by the API function `CxSetSAccelTimes()` are compared to the ramp time set by the `CxSetTrapAccelTimes()`. The S transition times will be compared to the ramp time and if they exceed the ramp time, the ramp time will be increased to the S times. In this case no linear transition will exist.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is not CX_TRAP_ACCEL or CX_S_ACCEL.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following program opens a mechanism for `CX_CONTROL`. It then sets the trajectory accel type to `CX_TRAP_ACCEL` for the first move, then changes it to `CX_S_ACCEL` for the second move.

```c
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    long      intrp_type, traj_accel_type;
    CxNodeId   MyMech, tcf, t1, t2;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );
    CxGetNamedNodeId( Server, "t2", &t2 );

    mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );

    CxSetTrajAccelType( mech, CX_TRAP_ACCEL );
    CxMoveToNode( mech, t1, tcf );
    CxSetTrajAccelType( mech, CX_S_ACCEL );
    CxMoveToNode( mech, t2, tcf );

    CxWaitForEndOfMotion( mech );
    CxRobpacExit();
}

SEE ALSO
CxGetTrajAccelType
**CxSetTrajectoryRate**

Sets the desired trajectory rate for path following

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetTrajectoryRate(CxMechanism mech, double rate)
```

**ARGUMENTS**

- **mech**: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **rate**: Trajectory rate, in seconds.

**DESCRIPTION**

This function allows the user to set the desired trajectory rate at which the mechanism’s kinematics are calculated. Mechanisms typically use values of 0.1 seconds (10 Hz) to 0.01 seconds (100 Hz), depending on the mechanism’s kinematic complexity — the lower the rate (higher frequency), the more accurate the path following. However, the amount of CPU usage also increases with a lower trajectory rate (see **WARNINGS** below). The default trajectory rate is 0.02 (50Hz).

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Value entered is less than 0.001.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**WARNINGS**

The user must be extremely cautious when setting these values if they will be used to actually control a mechanism. If the speed of the CODE kinematic calculations (forward and inverse) cannot be completed in the specified time interval, then the data sent to the physical controller will lag the rates expected by the hardware. This could cause unpredictable motion responses from the mechanism and other mechanisms being controlled, which could damage the mechanisms or surrounding equipment. The cautious user should first enter settings that are high (lower frequency) and then lower them slowly until an acceptable performance level is achieved with all processes running.

**SEE ALSO**

CxGetTrajectoryRate
**CxSetTrapAccelRamps**

Sets the desired rise and fall accelerations for curvilinear motion

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetTrapAccelRamps(CxMechanism mech, double rise_ramp, double fall_ramp)
```

**ARGUMENTS**

- `mech`  Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `rise_ramp`  Desired positive acceleration value in acceleration units (mm/s², in/s², etc.)
- `fall_ramp`  Desired positive deceleration value in acceleration units (mm/s², in/s², etc.)

**DESCRIPTION**

The function allows the user to set desired rise and fall acceleration rates for trajectory following in Cartesian space. If the desired value exceeds the maximums allowed for the mechanism, the desired values will internally be set to the maximums currently specified. The acceleration rates are only valid when the `CX_CONST_RAMP_ACCEL` acceleration type is in use (see `CxGetAccelType`, `CxSetAccelType`).

**RETURN VALUES**

This function returns `0` if successful; otherwise, `-1` (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CX_MESSAGE_SEND_FAILED</code></td>
<td>Error in sending message.</td>
</tr>
<tr>
<td><code>CX_MESSAGE_RECEIVE_FAILED</code></td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td><code>CX_INVALID_ARGUMENT</code></td>
<td>The acceleration entered is less than zero.</td>
</tr>
<tr>
<td><code>CX_MECH_NOT_OPEN_FOR_CONTROL</code></td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td><code>CX_INVALID_MECHANISM</code></td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first sets the motion related parameters. Next, it sets the velocity profile type to be of `CX_CONST_RAMP_ACCEL`. Finally, it gets the current settings for the acceleration and deceleration values and resets them if appropriate.

```c
#include <code/robpac.h>

#define MAXACCEL 0.1
#define MAXDECEL 0.1

void main(void)
{
    CxServer Server;
```
CxMechanism mech;
CxNodeId MyMech, tcf, t1, t2, t3;
double rise_time, fall_time;
double accel, decel;

Server = CxOpenServer( "Testing", CX_SMEM, 0 );
CxGetNamedNodeId( Server, "MyMech", &MyMech );
CxGetNamedNodeId( Server, "tcf", &tcf );
CxGetNamedNodeId( Server, "t1", &t1 );

mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO);
CxSetInterpType( mech, CX_LINEAR_INTERP );

/* Set motion velocity and acceleration profile parameters */
CxSetAccelType( mech, CX_CONST_RAMP_ACCEL );

/* Check velocity and acceleration profile parameters */
CxGetTrapAccelRamps( mech, &accel, &decel );
if ( (accel>MAXACCEL) || (decel> MAXDECEL) )
  CxSetTrapAccelRamps( mech, MAXACCEL*0.7, MAXDECEL*0.7 );
CxMoveToNode( mech, t1, tcf );

CxWaitForEndOfMotion( mech );
CxRobpacExit();

WARNINGS
The user must be extremely cautious when entering these values if they will be used to actually control a
mechanism since allowable accelerations are mechanism- and design-dependent. Excessive accelerations
may cause the mechanism’s inertial loads to damage the drive transmissions and other mechanical
components on the mechanism.

SEE ALSO
CxSetTrapAccelMax, CxGetTrapAccelRamps
**CxSetTrapAccelTimes**
Sets the desired rise and fall times for trapezoid motion profiles

**SYNOPSIS**
```
#include <code/robpac.h>
long CxSetTrapAccelTimes(CxMechanism mech, double rise_time, double fall_time)
```

**ARGUMENTS**
- `mech` Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- `rise_time` Acceleration time in seconds
- `fall_time` Deceleration time in seconds

**DESCRIPTION**
This function is used to set the current trapezoidal acceleration and deceleration times. These times are used if the mechanism’s current acceleration type is CX_CONST_RAMP_TIME (see CxGetAccelType, CxSetAccelType). These values may be set below the minimum accel times, but when the CIMServer calculates trajectories, the minimum accel times will be used (see CxSetAccelTimesMin).

When a CX_CONST_RAMP_TIME acceleration setting is used, S-curve acceleration and deceleration profiles can be combined with the constant trapezoidal acceleration and deceleration times respectively (see CxGetSAccelTimes, CxSetSAccelTimes). An S-curve acceleration profile is useful for minimizing the wear on a mechanism because the forces being applied to the mechanism’s gear train are changed gradually through the acceleration period.

The following illustrations show the relationship of the trapezoidal acceleration times and the S curve acceleration times. The figures include motion with no S-curve acceleration (pure trapezoidal acceleration profile), only S-curve acceleration, and combined S-curve and trapezoidal acceleration profiles. Note that in these figures, the acceleration profiles are the same as the deceleration profiles (i.e. the acceleration times are the same as the deceleration times); therefore, S_time is used to represent both. Likewise, rise_time and fall_time are used to represent trapezoidal acceleration times.

In order for the S-curve acceleration parameter to be used under CIMControl, the underlying motion control interface must support S-curve acceleration profiles (see HARDWARE AND SYSTEM DEPENDENCIES below).
NOTE: If the specified value for rise_S_time is greater than \( \frac{1}{2} \) the rise_time, the rise_S_time will be set to \( \frac{1}{2} \) the rise_time.

FUNCTIONALITY CHANGES

The functionality of CxSetTrapAccelTimes API has been changed in CODE v.3.6.4 release. This release note will explain how it was changed and will illustrate how to use it properly.

Originally, CxSetTrapAccelTimes was designed to allow the user to set the acceleration time and deceleration time explicitly to ensure all the joints finish acceleration phases and deceleration phases simultaneously for all joint interpolated moves.

This implementation causes the acceleration and deceleration values to change proportionally with the commanded velocity. If a user wishes to maintain the same acceleration values and deceleration values for each move regardless of the commanded velocity, he/she has to change the acceleration time and deceleration time proportionally to the commanded velocity.

The new implementation will allow the user to set the constant acceleration and deceleration values based on the maximum joint velocities. For all the subsequent joint interpolated moves, the dominant joint will be moving at commanded velocity with acceleration value and deceleration value calculated from the user specified acceleration time and deceleration time. The actual acceleration and deceleration times will be scaled down proportional to the commanded velocity. The algorithm for the new implementation will be outlined in the following steps.

For given acceleration and deceleration times, \( t_a \) and \( t_d \), CIMServer, either CIMulation or CIMControl, will use the maximum joint speed for each joint to find constant joint accelerations and joint decelerations.

\[
a_1 = \frac{V_{\text{max}1}}{t_a} \quad d_1 = \frac{V_{\text{max}1}}{t_d}
\]

If the joint speed is set to \( x\% \) of the maximum joint speed, the dominant joint, i.e., the joint that takes the longest time to move, will move at \( x\% \) of the maximum joint speed as specified. The actual acceleration time and deceleration time will be calculated based on the speed of the dominant joint.

\[
t_{a\text{-actual}} = x\% \times \frac{V_{\text{max}}}{a_1} = x\% \times t_a
\]
\[
t_{d\text{-actual}} = x\% \times \frac{V_{\text{max}}}{d_1} = x\% \times t_d
\]

The other joints will synchronize their motion with the dominant joint. These joints will move at a slower speed and lower acceleration value and deceleration value. The following diagrams will illustrate the difference between the two implementations:
RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Time entered is less than 0.005 sec.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program first sets the motion related parameters and sets the velocity profile type to be of
CX_CONST_RAMP_TIME. Then it gets the current settings for the rise and fall times and resets them if
appropriate.

```c
#include <code/robpac.h>
#define MIN_RISE_TIME 1.0
#define MIN_FALL_TIME 1.0

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   MyMech, tcf, t1, t2, t3;
    double    rise_time, fall_time;
    double    accel, decel;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "MyMech", &MyMech );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    CxGetNamedNodeId( Server, "t1", &t1 );
    ...
```
. mech = CxOpenMechanism( Server, MyMech, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO);
    CxSetInterpType( mech, CX_LINEAR_INTERP );

    /* Set motion velocity and acceleration profile parameters */
    CxSetAccelType( mech, CX_CONST_RAMP_TIME );

    /* Check velocity and acceleration profile parameters */
    CxGetTrapAccelTimes( mech, &rise_time, &fall_time );
    if ( (rise_time<MIN_RISE_TIME) || (fall_time<MIN_FALL_TIME) )
        CxSetTrapAccelTimes(mech, MIN_RISE_TIME*1.5,
                           MIN_FALL_TIME*1.5 );

    CxMoveToNode( mech, t1, tcf );
    .
    .
    CxWaitForEndOfMotion( mech );
    CxRobpacExit();
}

WARNINGS
The user must be extremely cautious when entering these values if they will be used to actually control a mechanism since allowable accelerations are mechanism- and design-dependent. Excessive accelerations may cause the mechanism’s inertial loads to damage the drive transmissions and other mechanical components on the mechanism.

HARDWARE AND SYSTEM DEPENDENCIES
On the MEI-DSP motion card, the acceleration profile is always the same as the deceleration profile. In this case only the rise_time parameter is used, and the fall_time parameter is ignored.

SEE ALSO
CxGetTrapAccelTimes, CxGetSAccelTimes, CxSetSAccelTimes, CxGetAccelTimesMin, CxSetAccelTimesMin
**CxSetTrapScrewAccel**

Sets the magnitude for trapezoidal rotational accels and decels

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetTrapScrewAccel(CxMechanism mech, double accel, double decel)
```

**ARGUMENTS**

- **mech**  
  Mechanism ID for mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.

- **accel**  
  Screw acceleration magnitude in rotational units (deg/ sec^2 or rad/ sec^2, depending on the units specified via `CxSetUnit`).

- **decel**  
  Screw deceleration magnitude in rotational units (deg/s sec^2 or rad/sec^2, depending on the units specified via `CxSetUnit`).

**DESCRIPTION**

This function sets the current screw acceleration and deceleration rates for Cartesian motions. When the acceleration type is `CX_CONST_RAMP_ACCEL` (see `CxGetAccelType`, `CxSetAccelType`), these magnitudes determine the acceleration and deceleration during linear and circular interpolated moves if the screw speed dominates the tool control frame (TCF) motion. If the current values are greater than the maximum rotational acceleration and deceleration magnitudes, then the values will be set to these maximums.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERROR**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Rising and falling accelerations must be greater than 0.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first sets the acceleration type to `CX_CONST_RAMP_ACCEL`. Next, it gets the value of trap screw acceleration and prints it out. Finally, it sets the new acceleration and deceleration values, reads them back and prints them out.

```c
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
```
Server = CxOpenServer( "Testing", CX_SMEM, 0 );
mech = CxOpenMechanism( Server, s100, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_WAIT);

/* Set motion parameters */
CxSetAccelType( mech, CX_CONST_RAMP_ACCEL );
CxGetTrapScrewAccel( mech, &current_accel, &current_decel);
printf(" Acceleration : %lf\t Deceleration : %lf \n",
       current_accel, current_decel);
CxSetTrapScrewAccel( mech, 1.0, 1.0 );
CxGetTrapScrewAccel( mech, &current_accel, &current_decel);
printf(" Acceleration : %lf\t Deceleration : %lf \n",
       current_accel, current_decel);

CxRobpacExit();
}

**WARNINGS**

The user must be extremely cautious when entering these values if they will be used to actually control a mechanism since allowable accelerations are mechanism- and design-dependent. Excessive accelerations may cause the mechanism's inertial loads to damage the drive transmissions and other mechanical components on the mechanism.

**SEE ALSO**

CxGetTrapScrewAccel, CxGetAccelType, CxSetAccelType, CxGetScrewAccelMax, CxSetScrewAccelMax
**CxSetWeaveOnOff**

Turns weaving on or off

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetWeaveOnOff(CxMechanism mech, long on_or_off)
```

**ARGUMENTS**

- `mech` Mechanism ID for mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- `on_or_off` Flag to enable or disable weaving function.

**DESCRIPTION**

This function sets the current weaving status to `CX_ON` or `CX_OFF`.

**RETURN VALUES**

This function returns 0 (CX_OK) if successful; otherwise, -1 (CX_ERROR) is returned.

**ERROR**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_WEAVE_ON_OFF_VALUE</td>
<td>Only CX_ON or CX_OFF are valid parameter values.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program sets the weave on or off setting to `CX_ON`.

```c
#include <stdio.h>
#include <code/robpac.h>

void main ( void )
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
    mech = CxOpenMechanism ( Server, s100, CX_CONTROL ) ;

    /* Set weave type */
    CxSetWeaveOnOff( mech, CX_ON );
}
```
.  
  CxRobpacExit ( ) ;  
}  

WARNINGS  
  Excessive use of weaving may damage the distal joints of the mechanism.  

SEE ALSO  
  CxSetWeaveType, CxGetWeaveOnOff
**CxSetWeaveType**

Sets the weave type, weave amplitude and weave frequency

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetWeaveType(CxMechanism mech, long type, double ampl, double freq)
```

**ARGUMENTS**

- **mech** Mechanism ID for mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.
- **type** Specifies the desired weave pattern.
- **ampl** Amplitude in current length units (>0).
- **freq** Frequency in weave oscillation (>0).

**DESCRIPTION**

This function sets the weave pattern, the weave amplitude and the weave frequency of an oscillation pattern that is superimposed on the normal path notion of the tool TCF. Weaving is typically used in welding where the groove is large enough to require substantial filler material.

The available patterns are `CX_SINE_WEAVE`, `CX_V_WEAVE`, `CX_Z_WEAVE`, `CX_DELTA_WEAVE`, `CX_CIRCLE_WEAVE`, and `CX_BOX_WEAVE`, as shown in the following figure.

![Weave Patterns](image)

The weave amplitude that the user enters should be interpreted as one-half the total distance of oscillation of the tool. The frequency is the number of times the weave pattern will be repeated in one second.
Only mechanisms which have terminal joints that are rotational and which intersect, can be used in weld weaving. CODE will return an error if the mechanism joints are not compatible for weld weaving. The trajectory rate used for the mechanism must be at least 4 times greater than the weave frequency, else it may not be possible to properly discriminate the weave pattern shape. For most practical 5 and 6 axis mechanisms, the weave frequency will probably lie in the range of 1-10 Hz.

The weld torch TCF should not be set too close to the joint axes, or the oscillation may cause the joint motion to exceed their speed limits. This will result in the mechanism slowing down. The user can use CODE simulation to determine whether this may occur. The user can use the desired amplitude, frequency, and the approximate radial distance of the weld torch TCF from the joint origin to calculate the required joint speed and then compare this value to those of the terminal joints.

Note that this API function only sets the weave parameters. It requires the API function CxSetWeaveOnOff() to actually turn weaving on or off.

RETURN VALUES
This function returns 0 (CX_OK) if successful; otherwise, -1 (CX_ERROR) is returned.

ERROR
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism was not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>Parameter values must be greater than 0.</td>
</tr>
<tr>
<td>CX_WRONG_TYPE_WEAVE_JOINTS</td>
<td>Distal joints must be revolute and intersect.</td>
</tr>
<tr>
<td>CX_LOWER_WEAVE_FREQ</td>
<td>Must lower the weave frequency</td>
</tr>
<tr>
<td>CX_TOOL_TOO_CLOSE_TO_WEAVE_JOINTS</td>
<td>Will exceed joint speeds unless tool moved further away</td>
</tr>
<tr>
<td>CX_TOOL_LESS_THAN_MIN_WELD_ANGLE</td>
<td>Adjust torch Z axis so that torch is greater than minimum weld angle.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program sets the weave type to CX_SINE_WEAVE, the weave amplitude to 5 mm and the weave frequency to 5 Hz.

```c
#include <stdio.h>
#include <code/robpac.h>

void main( void )
{
    CxServer Server;
    CxMechanism mech;
    CxNodeId   s100;

    Server = CxOpenServer ( "Testing", CX_SMEM, 0 ) ;
```
mech = CxOpenMechanism (Server, s100, CX_CONTROL);
/* Set weave type */
CxSetWeaveType (mech, CX_SINE_WEAVE, 5.0, 5.0);
.
CxRobpacExit();
}

WARNINGS

The user must be cautious when entering these values if they will be used to actually control a mechanism since weaving might not be a capability intended for the mechanism. Excessive use of weaving may damage the distal joints of the mechanism.

SEE ALSO

CxGetWeaveType, CxSetWeaveOnOff, CxGetWeaveOnOff
**CxTeachConfig**

Teaches a configuration

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxTeachConfig(CxMechanism mech, char *name, long over_write, long serve_type)
```

**ARGUMENTS**

- **mech**: Mechanism ID for mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
- **name**: Configuration name
- **over_write**: Overwrite flag; when set to CX_TRUE, the new configuration overwrites an existing configuration of the same name (CX_TRUE and CX_FALSE)
- **serve_type**: Server mode (CX_ANIMATION or CX_RUNTIME)

**DESCRIPTION**

This function teaches a configuration associated with the specified mechanism. A configuration is defined by the mechanism’s current joint positions. A configuration can be taught in CX_ANIMATION mode or CX_RUNTIME mode. The name argument associates a symbolic name with this configuration. If it is desirable to return to the pre-taught configuration, CxMoveToConfig can be used.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_CONFIG_EXISTS</td>
<td>Configuration already exists and overwrite flag is set to CX_FALSE.</td>
</tr>
<tr>
<td>CX_MACHINE_OUT_OF_MEMORY</td>
<td>Memory allocation failed.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program first creates a 3-D path, then saves the current configuration. Next it moves a TCF along the path. It then moves the mechanism back to the original configuration. Finally, it deletes the configuration.

```c
#include <stdio.h>
```
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId    s100, path, tcf;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "s100", &s100 );
    CxGetNamedNodeId( Server, "path", &path );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );
    CxSetBlendPolicy( mech, CX_MOVE_TO);

    /* Create a curve to be used for the tool movement */
    CxAddCurveSeg( Server, path, "", "pt1", "XYZ", 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0 );
    CxAddCurveSeg( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
                   0.0, 140.0, 500.0 );

    CxTeachConfig( mech, "orig_config", CX_TRUE, CX_ANIMATION );

    CxMoveRelPath( mech, path, tcf, "", "", "XYZ", 0.0, 0.0, 0.0,
                  5.0, 5.0, 5.0 );

    CxMoveToConfig( mech, "orig_config" );

    CxDeleteConfig( mech, "orig_config" );

    CxRobpacExit();
}

SEE ALSO
    CxMoveToConfig, CxDeleteConfig, CxGetConfigName, CxGetNumConfig,
    CxListConfig
CxTeachNode

Creates a node and teaches its pose using a TCF

SYNOPSIS

#include <code/robpac.h>
long CxTeachNode(CxMechanism mech, char *name, CxNodeId parent, CxNodeId tcf)

ARGUMENTS

mech Mechanism ID for mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.
name Node name
parent The parent node ID of the new node
tcf The TCF node ID of a TCF attached to the mechanism

DESCRIPTION

This function creates a node as a child of a parent node. Its pose with respect to its parent is set such that the new node will be at same physical position and orientation as the specified TCF node. In other words, the new node will be posed to exactly coincide with the TCF node.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_PARENT_NODE_NOT_FOUND</td>
<td>Invalid parent node ID.</td>
</tr>
<tr>
<td>CX_NODE_EXISTS</td>
<td>Named node already exists under parent.</td>
</tr>
<tr>
<td>CX_TCF_NOT_FOUND</td>
<td>Invalid TCF node ID.</td>
</tr>
<tr>
<td>CX_MACHINE_OUT_OF_MEMORY</td>
<td>Memory allocation failed.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_PARENT_NOT_FOUND</td>
<td>Specified parent node does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program creates a 3-D path, then creates a teach node. The teach node provides a reference point so the TCF can be moved to this position anytime in the future. The program then moves the TCF along the path, and back to the teach point.
#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100, path, tcf, org_pos;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );
    CxGetNamedNodeId( Server, "s100", &s100 );
    CxGetNamedNodeId( Server, "path", &path );
    CxGetNamedNodeId( Server, "tcf", &tcf );
    mech = CxOpenMechanism( Server, s100, CX_CONTROL );

    /* Create a curve to be used for the tool movement */
    CxAddCurveSeg( Server, path, "", "pt1", "XYZ", 0.0, 0.0, 0.0,
                  0.0, 0.0, 0.0 );
    CxAddCurveSeg( Server, path, "pt1", "pt2", "XYZ", 45.0, 0.0, 0.0,
                  0.0, 140.0, 500.0 );
    ... 
    CxTeachNode( mech, "org_position", world, tcf);
    CxGetNamedNodeId ( Server, "org_position", &org_pos );

    CxMoveRelPath( mech, path, tcf, "", "", "XYZ", 0.0, 0.0, 0.0
                  5.0, 5.0, 5.0 );
    CxMoveToNode ( mech, org_pos, tcf );
    ... 
    CxRobpacExit();
}

SEE ALSO

CxReteachNode,CxMoveRelNode
CxUpdateJoints

Updates mechanism joint values in a database with actual mechanism joint values

SYNOPSIS

```c
#include <code/robpac.h>
long CxUpdateJoints(CxMechanism mech)
```

ARGUMENTS

- `mech` - Mechanism ID of mechanism open for control. This is a `CxMechanism` ID returned by a call to `CxOpenMechanism`.

DESCRIPTION

This function updates the mechanism joint values in the database after the mechanism has been jogged or moved. This function helps establish the correct relationship between the ideal (simulated) mechanism and the actual mechanism.

If the mechanism’s servo amplifiers are enabled, this request is placed on the CIMServer’s motion queue, and will not be processed until all prior motion related requests have completed execution. If the amplifiers are disabled, this function updates the mechanism’s joint values immediately and is not processed through the motion queue.

RETURN VALUES

This function returns 0 if successful; -1 (CX_ERROR) if an error occurs; and 1 (CX_GOT_SIGNAL) if the process had previously registered an interrupt handler (such as `interrupt_on_value`) with the mechanism specified. It will also produce a 1 if a signal interrupt occurs, and a previously queued motion was aborted using `CxSendMechanismErrorAction` in the signal handler.

ERRORS

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX.MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX.MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX.INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following code will do rigid body updating for a part with reference frame `ref`. A vision camera is used to measure the rigid body deviations.

```c
CxMechanism mech;
double mspeed = 1.0;
double aspeed = 0.1;
double apprch = 10.0;
Sensor *sensor;
CxNodeId ref;
char axes[4];
CxVector ang, vec;
```
fprintf(stderr,"Rigid body correction on the part.\n");

/* set joint speed to move_speed */
CxSetJointSpeed( mech, mspeed );

/* move camera (sensor->snsr_id) to ref with offset approach */
CxMoveRelNode( mech, ref, sensor->snsr_id, "zyx",
              0.,0.,0.,0.,0.,apprch );

/* set joint speed to approach_speed */
CxSetJointSpeed( mech, aspeed );

/* move the camera frame to ref frame on part */
CxMoveRelNode( mech, ref, sensor->snsr_id, "zyx",
              0., 0., 0., 0., 0., 0. );

/* update the mechanism joint values */
CxUpdateJoints( mech );

/* measure rigid body deviations with measurement method #1 */
/* measure_pose is a custom function*/
measure_pose( server, ref, sensor, 1, axes, ang, vec );

/* update the rigid body where ref frame is attached */
CxSetPose( mech, ref, sensor->snsr_id, ref, axes, ang, vec );

/* save the updated state as default state */
CxSaveState( Server, "default.st" );

/* set joint speed to move_speed */
CxSetJointSpeed( mech, mspeed );

SEE ALSO

CxJogRobot, CxSendMechanismErrorAction, CxSetPose
CxUpdateTcf

Updates TCF (tool control frame) in the database

SYNOPSIS
#include <code/robpac.h>
long CxUpdateTcf(CxMechanism mech, CxNodeId tcf, CxNodeId update_ele,
    char axes[4], CxVector angles, CxVector vec)

ARGUMENTS

mech     Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by
         a call to CxOpenMechanism.

tcf      Node ID of frame to be updated. This is the value returned by a call to
         CxGetNamedNodeId.

update_ele Node ID of frame to be updated. This is the value returned by a call to
            CxGetNamedNodeId.

axes     Order of rotation for angles (e.g. XYZ).

angles   Relative rotation angles (in degrees unless the unit is set to CX_USE_RADIAN. See
         CxSetUnit in the Nodes, Frames, and Attributes section of the CODE API

vec      Relative position vector.

NOTE: Type CxVector is defined in <code/matx_def.h> as follows:
        typedef double CxVector[3];

DESCRIPTION

This function updates a tool TCF in the database with its calibrated pose, a better approximation of where the
actual tool is with respect to its parent frame (usually the tool interface frame or TIF).

The TCF can also be updated with any other frame (node passed in as update_ele) above the TCF frame,
so that the updated TCF in the database will be a true representation of the actual TCF. The parameters axes,
angles, and vec represent the displacement of the TCF with respect to the mechanism's last joint frame.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_MEC_MH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
<tr>
<td>CX_NON_XYZ_AXIS</td>
<td>An illegal character was used in the axes string.</td>
</tr>
<tr>
<td>CX_NODE_NOT_FOUND</td>
<td>Given node does not exist.</td>
</tr>
</tbody>
</table>
**Error Code**  | **Description**  
---|---
CX_INVALID_NODE_ID  | Node number does not match node index number.  
CX_NODE_IS_CUT_OUT  | Given node has been cut out.  

**EXAMPLE**

The following code segment uses TCF pose values obtained from automated TCF calibration procedures (left out for clarity and brevity) to update the ideal (workcell file) TCF pose to match the real, calibrated pose of the physical TCF, using CxUpdateTcf. The workcell state, including the updated TCF pose, is then saved with the CxSaveState function from the “Browser and Tree” library.

```c
CxMechanism mech;
CxNodeId vac_tcf;
CxVector ang, vec;
char axes[4];
double THETA, TCF_X, TCF_Y, TCF_Z;

/* TCF values from automated TCF calibration process: 
   THETA, TCF_X, TCF_Y, TCF_Z */
ang[0] = THETA;
ang[1] = ang[2] = 0.0;
vec[0] = TCF_X;
vec[1] = TCF_Y;
vec[2] = TCF_Z;
strcpy(axes, "ZYX");

/* update TCF in the workcell database */
CxUpdateTcf( mech, vac_tcf, vac_tcf, axes, ang, vec);

/* save this updated workcell state */
CxSaveState(Server, "default.st");
```

**SEE ALSO**

CxUpdatePose, CxSetPose, CxSetUnit, CxGetUnit
CxWaitForEndOfCommandedMotion

Waits for CIMControl to complete all the moves accumulated in the queue without waiting for the mechanism axes to settle.

SYNOPSIS

```
#include <code/robpac.h>
long CxWaitForEndOfCommandedMotion (CxMechanism mech)
```

ARGUMENTS

- `mech`: Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION

This function halts the execution of the client process until the server finishes all the move commands that have accumulated in the motion queue prior to this statement. If the `blend_policy` is `CX_MOVE_WAIT`, this function will return immediately, since all motion will be complete by the time this function is called.

This function differs from CxWaitForEndOfMotion in that it does not wait for the mechanism axes to settle into their commanded positions.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
CxWaitForEndOfMotion
Blocks for the server to complete all the moves

SYNOPSIS
#include <code/robpac.h>
long CxWaitForEndOfMotion (CxMechanism mech)

ARGUMENTS
mech Mechanism ID of mechanism open for control. This is a CxMechanism ID returned by a call to CxOpenMechanism.

DESCRIPTION
This function halts the execution of the client process until the server finishes all the move commands that have accumulated in the motion queue prior to this statement. If the blend_policy is CX_MOVE_WAIT, this function will return immediately, since all motion will be completed by the time this function is called.

RETURN VALUES
This function returns 0 if the move completes successfully; -1 if an error occurs; or 1 if the process had previously registered an interrupt handler (such as interrupt_on_value) with the mechanism specified. It will also produce a 1 if a signal interrupt occurs, and a previously queued motion was aborted using CxSendMechanismErrorAction in the signal handler.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Cannot send message to the given server.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Cannot receive returned message from the given server.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not opened for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>Invalid mechanism ID, or mechanism does not exist.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program opens a mechanism, sets the blend policy to CX_MOVE_TO and issues commands to move joints. It moves axis #0 to a joint value of 100.0 and axis #1 to a joint value of 120.0. As the blend policy is set to CX_MOVE_TO, function calls will return even if the corresponding moves are incomplete. To avoid calling function CxRobpacExit before all the moves are completed, CxWaitForEndOfMotion is called.

#include <stdio.h>
#include <code/robpac.h>

void main(void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
}
Server = CxOpenServer( "Testing", CX_SMEM, 0 );
mult = CxOpenMechanism( Server, s100, CX_CONTROL );
CxSetBlendPolicy( mech, CX_MOVE_TO );

CxMoveSingleAxis( mech, 0, 100.0 );
CxMoveSingleAxis( mech, 1, 120.0 );
CxWaitForEndOfMotion( mech );
CxRobpacExit();
}
Events, States, and I/O
CxAsyncPulseSignal
Asynchronously changes a signal’s value for a specified length of time

SYNOPSIS
#include <code/robpac.h>
long CxAsyncPulseSignal (CxServer Server, long signal_num, long value,
long time)

ARGUMENTS
Server Server ID
Signal_num Signal number
value New value for the signal
time Time in milliseconds to pulse the signal

DESCRIPTION
This function sets a signal’s value for a specified amount of time, then resets the signal's value back to the
state it had before CxAsyncPulseSignal was called.

If the CIMServer is in CX_RUNTIME mode, the signal must be a CX_SOFTWARE or an output signal; otherwise, an error will occur. If the Server is in CX_ANIMATION mode, any signal can be pulsed.

This function starts the pulsing of a signal but does not wait for the pulse to complete. If the return should be
delayed until after the signal has been pulsed, then the CxPulseSignal function should be used.

If the signal value is changed before the pulse time is up, that change will not remain in effect after the signal
changes back to its original state.

When the server is in CX_ANIMATION mode, this function is the same as CxPulseSignal. The function
works as described for the threaded version of the CIMServer operating in CX_RUNTIME mode; otherwise, it
works just as CxPulseSignal does.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The specified signal is not an output signal, and the Server is in CX_RUNTIME mode.</td>
</tr>
<tr>
<td>CANT_CREATE_PULSE_THREAD</td>
<td>The system could not spawn a thread for pulsing the desired signal.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following program reads the value of a signal before entering a loop which continuously monitors the signal value. Since the pulse was set to be a synchronous pulse, the program does not continue execution until the pulse is over. When the program does resume checking, the pulse signal value is always high; the program never enters the first while loop.

The program then waits one second and starts an asynchronous pulse which lasts for 5 seconds. This time, control goes back to the program immediately and the program continuously polls the signal status until the pulse is over.

NOTE: PULSE_SIGNAL is assumed to be a signal table index defined in sigTable.h.

```c
#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h"
#define PULSE_HIGH 1
#define PULSE_LOW 0

void main ( void )
{
    CxServer    Server;
    long      pulse_state;
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    /* Start and wait till the first pulse is over */
    CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
    CxPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
    CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
    while (pulse_state != PULSE_HIGH)
    {
        /* This portion will never get executed */
        printf(" Pulse is still not over \n");
        CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
    }
    printf(" The first pulse signal has completed \n");
    CxDelay(1,0);

    /* Start and wait till the second pulse is over */
    CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
    CxAsyncPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
    CxGetSignal(Server, PULSE_SIGNAL,);
    while (pulse_state != PULSE_HIGH);
    {
        printf(" Pulse is still not over \n");
        CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
    }
    printf(" The second pulse signal has completed \n");
    CxRobpacExit();
}
```

SEE ALSO

CxPulseSignal
CxDelay
Waits for a given number of seconds and microseconds

SYNOPSIS
#include <cimUtils/basicUtils.h>
void CxDelay(long sec, long microsec)

ARGUMENTS
sec Number of seconds to delay
microsec Number of microseconds to delay (1 microsecond = 10^{-6} second)

DESCRIPTION
This function is used to cause a time delay of a given number of seconds and microseconds.

In the Windows NT environment, the default delay granularity is typically 10 milliseconds. This can
be controlled using the functions timeBeginPeriod and timeEndPeriod. See the Microsoft Visual C++
Multimedia programmer’s reference to learn how to use these functions.

EXAMPLE
The following program reads the value of a signal before entering a loop which continuously monitors the
signal value. Since the pulse was set to be a synchronous pulse, the program does not continue execution until
the pulse is over. When the program does resume checking, the pulse signal value is always high; the program
never enters the first while loop.

The program then waits one second and starts an asynchronous pulse which lasts for 5 seconds. This time,
control goes back to the program immediately and the program continuously polls the signal status until the
pulse is over.

NOTE: PULSE_SIGNAL is assumed to be a signal table index defined in sigTable.h.

#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h"
#define PULSE_HIGH 1
#define PULSE_LOW 0

void main ( void )
{
    CxServer Server;
    long pulse_state;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    /* Start and wait till the first pulse is over */
    CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
    CxPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
    CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
    while (pulse_state != PULSE_HIGH)
    {
        /* This portion will never get executed */
        printf(" Pulse is still not over \n");
        CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
    }
    printf(" The first pulse signal is over \n");
CxDelay(1,0);

/* Start and wait till the second pulse is over */
CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
CxAsyncPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
CxGetSignal(Server, PULSE_SIGNAL);
while (pulse_state != PULSE_HIGH);
{
    printf(" Pulse is still not over \n");
    CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
}
printf(" The second pulse signal is over \n");

CxRobpacExit();

SEE ALSO

The CODE Application Programming manual.
**CxDisablePosComp**

Disables the position comparing function

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxDisablePosComp(CxMechanism mech, long axis)
```

**ARGUMENTS**

- **mech** mechanism ID
- **axis** the axis whose position is being compared

**DESCRIPTION**

This function disables the position compare function for an axis.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 is returned.

**ERRORS**

If the function returns -1(CX_ERROR), the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_INVALD_ARGUMENT</td>
<td>The input values are invalid arguments in the function definition.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

```c
#include <stdio.h>
#include <code/robpac.h>
#include <code/robconst.h>

#define  POS_COMP_TRIG_1     46

void main( void )
{
    double compPos[CX_MAX_POS_COMP];
    double rollover;
    CxServer Server;
    CxMechanism Mech;

    /* Open CxServer and CxMechanism */

    .
    .
    .
}
```
rollover = 360.0;
compPos[0] = 60.0;
compPos[1] = 120.0;
compPos[2] = 180.0;
compPos[3] = 240.0;

CxPositionComp( Mech, 0, 4, compPos, POS_COMP_TRIG_1, 1, rollover );

CxMoveSingleAxis( Mech, 0, 3600.0 );

CxDisablePosComp( Mech, 0 );

HARDWARE AND SYSTEM DEPENDENCIES

At present, this function is only supported for use with the PMAC and MEI DSP motion cards.

SEE ALSO

CxPositionComp
CxDisableWhen

Disables a previous call to CxWhen

SYNOPSIS

#include <code/robpac.h>
long CxDisableWhen (CxServer Server, long when_signal, long when_value,  
                    long CxSetSignal, long set_value);

ARGUMENTS

Server        Server ID
when_signal   Input signal (number) being monitored
when_value    Value that the input signal must change to in order to trigger the action
set_signal    Output signal (number) that is to be set when the conditions are met
set_value     Value that the output signal is to be set to when the conditions are met

DESCRIPTION

The CxWhen function allows an application process to register a signal output action to be taken when the  
value of a specified input signal changes to a given state. Once the action has been registered, the server  
assumes the responsibility of monitoring the input signal and setting the specified output, thus relieving the  
application process of the burden of polling the input directly.

The specified action is executed each time the input signal changes to the given state. Once the action has  
been registered, it remains in effect until the application process unregisters it by calling CxDisableWhen,  
or until the process exits.

NOTE: To disable the when condition, all of the signal parameters must be exactly the same as those used to  
register the when condition; otherwise, an error will occur.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber  
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_NO_SUCH_ENTRY</td>
<td>A corresponding when function has not been called.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following example causes the REDLIGHT output signal to go on whenever the BADPART input signal  
changes to CX_TRUE. This condition is monitored until the CxDisableWhen function is called.

.

.


{  
  CxServer myServer;  
  .  
  /* Enable the when condition */  
  CxWhen(myServer, BADPART, CX_TRUE, REDLIGHT, CX_ON);  
  .  
  /* Disable the when condition */  
  CxDisableWhen(myServer, BADPART, CX_TRUE, REDLIGHT, CX_ON);  
  .  
}

SEE ALSO

CxWhen
**CxGetSignalId**

Gets the numerical index of a named signal

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxGetSignalId(CxServer Server, char *name, long *index)
```

**ARGUMENTS**

- **Server**: Server ID
- **name**: Logical signal name
- **index**: The index into the signal table corresponding to the signal name

**DESCRIPTION**

This function retrieves the numerical index of a specific I/O signal in the CIMServer’s I/O signal table of a specific I/O signal given its logical name. This function is useful in cases where the signal table has been modified, and the user does not wish to change the application executable.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal name is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following example obtains the index of the CYCLE_CONVEYOR I/O signal, and then sets that signal.

```c

/* Open Server */

/* Get Signal ID */
CxGetSignalId ( myServer, "CYCLE_CONVEYOR", &conveyor_sig_id ) ;
CxSetSignal ( myServer, conveyor_sig_id, CX_TRUE ) ;
```

**SEE ALSO**

CxSetSignalName
CxGetSignalName

Gets the name of a signal using the signal ID

SYNOPSIS
#include <code/robpac.h>
long CxGetSignalName(CxServer Server, long index, char *name)

ARGUMENTS
Server  Server ID
index   The index into the signal table.
name    Logical signal name corresponding to the signal index

DESCRIPTION
This function determines the logical name of an I/O signal defined in the CIMServer’s signal table, given the corresponding numerical index into the table.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal name is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following example determines the name of the signal with index defined as CYCLE_CONVEYOR.

#include <code/robpac.h>
#include "mySigDefs.h"  /* Contains definition of CYCLE_CONVEYOR */
.
.
CxServer myServer;
long conveyor_sig_id;
char signame[50];

/* Open Server */
.
.
/* Get Signal name */
CxGetSignalName ( myServer, CYCLE_CONVEYOR, signame );
printf ( "Signal %ld has name %s\n", CYCLE_CONVEYOR, signame );
}
SEE ALSO

CxSetSignalId
CxGetSignalValue

Gets the value of a specified signal

SYNOPSIS

```c
#include <code/robpac.h>
long CxGetSignalValue(CxServer Server, long signal_num, long *value)
```

ARGUMENTS

- **server**: Server ID
- **signal_num**: Signal number
- **value**: The value of the given signal

DESCRIPTION

This function determines the current state of an input or output signal. For both input and output signals, the state of the signal is determined by querying the CIMServer’s internal signal table. One exception to this is when the following conditions are met:

1. The signal is an input signal
2. The server is in `CX_RUNTIME` mode, and
3. The signal is a hardware signal.

If the conditions are met, then the signal’s state is determined by querying its corresponding hardware.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program reads the value of a signal before entering a loop which continuously monitors the signal value. Since the pulse was set to be a synchronous pulse, the program does not continue execution until the pulse is over. When the program does resume checking, the pulse signal value is always high; the program never enters the first while loop.

The program then waits one second and starts an asynchronous pulse which lasts for 5 seconds. This time, control goes back to the program immediately and the program continuously polls the signal status until the pulse is over.

NOTE: **PULSE_SIGNAL** is assumed to be a signal table index defined in `sigTable.h`.

```c
#include <stdio.h>
#include <code/robpac.h>
```
#include "sigTable.h"

void main ( void )
{
    CxServer Server;
    long sig;

    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    CxGetSignalValued(Server, PULSE_SIGNAL, &sig)
    if (sig==1)
        printf("Signal is set \n");
    else if (sig==0)
        printf("Signal is cleared \n");
    else
        printf("Signal is in an unknown state \n");
    CxRobpacExit();
}

SEE ALSO

CxSetSignal
# CxInterruptOnChange

Sets up a signal handler for a specific signal on any value change

## SYNOPSIS

```c
#include <code/robpac.h>
long CxInterruptOnChange(CxServer server, long signal_num, CxMechanism mech, void (*handler Proc)())
```

## ARGUMENTS

- **server**
  - Server ID
- **signal_num**
  - Signal number
- **mech**
  - Mechanism ID. The mechanism will stop when the signal changes state. (Mechanism must be open for CX_CONTROL.) If no mechanism is involved, a CX_NULL pointer may be used.
- **sandler_procc**
  - User-supplied signal handler function

## DESCRIPTION

This function causes a user-supplied signal handling routine to be invoked whenever the specified signal changes state. Upon returning from the signal handler, the process resumes from the pre-interrupt location in the application. The handler stays active until a call to CxInterruptOnChangeOff is made.

The defined signal can be either an input or an output signal. A mechanism can also be halted when the specified signal changes state. This is done by passing a valid mechanism pointer, obtained using the CxOpenMechanism function through the mech argument.

Once the signal is triggered and the mechanism is halted, the process must specify the default action to be taken on the current motion and any other motions pending in the move queue on the CIMServer. This is done using the CxSendMechanismErrorAction function. The possible actions, which can be taken after this function has been invoked, are the following:

- Abort the current motion and flush the remaining moves in the queue,
- Ignore the rest of the current move and process the remaining moves in the queue,
- Resume the current move and finish processing the remaining moves in the queue, or

## NOTE:

If a subsequent call is made to CxInterruptOnChange with the same signal number, the previous mechanism and signal handler functions are overwritten by the new mechanism and signal handler function.

The signal handler routine should have the following function prototype format:

```c
void handler_procc( long signal, long value, CxMechanism mech )
```

The arguments for this routine are defined as follows:

- **signal**
  - The signal which triggered the interrupt
- **value**
  - The current state of the signal
- **mech**
  - Mechanism ID which was registered with the interrupt handler. Mechanism must be open for CX_CONTROL. A value of CX_NULL may be passed for this parameter meaning that no interaction with the mechanism is desired.

## RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.
ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer's signal table.</td>
</tr>
<tr>
<td>CX_NO_SIGNAL_FUNC_ROUTINE</td>
<td>The driver type specified in the signal table has not been implemented in the cntrl_signal_func function in the CIMServer.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>An invalid mechanism pointer was passed in through the mech argument.</td>
</tr>
</tbody>
</table>

EXAMPLES

This example causes the user function actuate_gripper to be invoked whenever signal GRIP_PART changes, the program simply sleeps at all other times:

```c
#include <stdio.h>
#include <code/robpac.h>
#include "sigdefs.h"

void actuate_gripper(long signal, long value, CxMechanism mech);
CxServer Server;
CxMechanism grip, mech ;
CxNodeId MyMech_node, grip_node ;
CxNodeId finger1, finger2;
.
.
void main (void)
{
    Server = CxOpenServer("gripper", CX_SMEM, 0);
    .
    /* Get node_id’s */
    CxGetNamedNodeId ( Server, "roby", &MyMech_node );
    CxGetNamedNodeId ( Server, "Gripper", &grip_node );
    CxGetNamedNodeId ( Server, "finger1", &finger1 );
    CxGetNamedNodeId ( Server, "finger2", &finger2 );
    .
    mech = CxOpenMechanism ( Server, MyMech_node,CX_CONTROL );
grip = CxOpenMechanism ( Server, grip_node,CX_CONTROL );
CxInterruptOnChange( Server, GRIP_PART, mech, actuate_gripper);
while(1)
    pause();
```
void actuate_gripper(long signal, long value, CxMechanism mech )
{
    if (value == CLOSE_HAND)
    {
        CxMoveSingleAxis(grip, 0, 50.0);
        CxChangeColor(Server, finger1, 0.0, 0.0, 1.0);
        CxChangeColor(Server, finger2, 0.0, 0.0, 1.0);
        CxSetSignal(Server, PART_GRIPPED, CX_TRUE);
    }
    else
    {
        CxMoveSingleAxis (grip, 0, 60.0);
        CxChangeColor(Server, finger1, 1.0, 0.0, 0.0);
        CxChangeColor(Server, finger2, 1.0, 0.0, 0.0);
        CxSetSignal(Server, PART_GRIPPED, CX_FALSE);
    }
    CxSendMechanismErrorAction ( mech, CX_MECH_RESUME ) ;
}

SEE ALSO

CxInterruptOnChangeOff, CxSendMechanismErrorAction
CxInterruptOnChangeOff
Disables a signal handler for a signal on change to any value

SYNOPSIS
#include <code/robpac.h>
long CxInterruptOnChangeOff(CxServer Server, long signal_num)

ARGUMENTS
Server Server ID
signal_num Signal number

DESCRIPTION
This function disables a signal interrupt handler previously registered using the CxInterruptOnChange function. Once called, the interrupt handler previously registered will no longer be invoked if the signal changes state.

Any mechanism specified in the corresponding CxInterruptOnChange function will no longer be halted if the signal changes state after this function is executed.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer's signal table.</td>
</tr>
<tr>
<td>CX_NO_SUCH_ENTRY</td>
<td>A CxInterruptOnChange handler has not been registered with the CIMServer for the given signal.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program sets a signal value to 0 and an interrupt handler for any change in the signal value. Next, it changes the value of the signal. Note that every time the signal value changes, the interrupt handler is called and a message is displayed. Then it turns off the interrupt handler. After that, changing a signal value does not trigger the interrupt handler. It is assumed that SW1_SIGNAL is defined in sigTable.h, which is found in the local directory.

```
#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h"  /* define signals and their states */
void value_changed(long signal, long value, CxMechanism mech);

void main (void)
{
```
CxServer Server;
Server = CxOpenServer( "Testing", CX_SMEM, 0 );

CxSetSignal( Server, SW1_SIGNAL, 0 );
CxInterruptOnChange( Server, SW1_SIGNAL, CX_NULL, value_changed);
CxSetSignal( Server, SW1_SIGNAL, 10 );
CxSetSignal( Server, SW1_SIGNAL, 20 );
CxInterruptOnChangeOff(Server, SW1_SIGNAL);
CxSetSignal( Server, SW1_SIGNAL, 10 );
CxSetSignal( Server, SW1_SIGNAL, 20 );
CxRobpacExit();

}

void value_changed(long signal, long value, CxMechanism mech)
{
    printf(" Change in signal value : %ld \n", value);
}

SEE ALSO

CxInterruptOnChange
**CxInterruptOnLowerThreshold**

Sets up a signal handler for lower threshold monitoring

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxInterruptOnLowerThreshold(CxServer Server, long signal_num, long value, CxMechanism mech, void (*handler_proc)())
```

**ARGUMENTS**

- **Server**  Server ID
- **signal_num**  Signal number
- **value**  Signal value below which an interrupt handler will be called
- **mech**  Mechanism ID. The mechanism will stop when the signal reaches or drops below the specified threshold. If no mechanism is included, a CX_NULL pointer may be used (the mechanism must be open for CX_CONTROL).
- **handler_proc**  User-supplied signal handler routine

**DESCRIPTION**

This function causes a user-supplied signal handling routine to be invoked whenever the value of the specified signal reaches or drops below the specified value. Upon return from the signal handler, the process resumes from the pre-interrupt location in the application.

The defined signal can be either an input or an output signal. A mechanism can be halted when the specified signal reaches its lower threshold state. This is done by passing a valid mechanism pointer, obtained using the `CxOpenMechanism` function, through the `mech` argument.

Once the signal is triggered, and the mechanism is halted, the process must specify the default action to be taken on the current motion, and any other motions pending in the move queue on the CIMServer. This is done using the `CxSendMechanismErrorAction` function. The possible actions which can be taken when this function is invoked are the following:

- Abort the current motion and flush the remaining moves in the queue,
- Ignore the rest of the current move and process the remaining moves in the queue,
- Resume the current move and finish processing the remaining moves in the queue, or

**NOTE:** If a subsequent call is made to `CxInterruptOnLowerThreshold` without disabling the previous call and with the same signal number and value, the previous mechanism and original handler functions are overwritten by the new mechanism and signal handler function.

The user’s handler is invoked when the value of the signal drops below the specified threshold. The handler stays active until it is disabled by calling the `CxInterruptOnLowerThresholdOff` API function. `CxInterruptOnLowerThreshold` is automatically disabled after the threshold is passed once. To use this function more than once, it must be reenabled in the interrupt handler routine.

If the signal is still in the specified range when this function is reset, the interrupt handler gets called immediately. To avoid possible infinite loops, wait for the signal value to go above the threshold level before resetting `CxInterruptOnLowerThreshold`.

The interrupt routine should have the following format and functional arguments:

```c
void handler_proc ( long signal, long value, CxMechanism mech )
```

The arguments to this routine are described as follows:
signal  The signal which triggered the interrupt
value  The current state of the signal
mech   Mechanism ID which was registered with the interrupt handler. Mechanism must be
open for CX_CONTROL. CX_NULL means that no interaction with the mechanism is
desired.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_NO_SIGNAL_FUNC_ROUTINE</td>
<td>The driver type specified in the signal table has not been implemented in the cntrl_signal_func function in the CIMServer.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>An invalid mechanism pointer was passed in through the mech argument.</td>
</tr>
</tbody>
</table>

EXAMPLE
This example causes the user function handler proc to be invoked whenever signal 10 drops below the value of 5. When this occurs, the mechanism is halted and all motions pending in the queue are aborted. The program simply sleeps at all other times.

```
#include <code/robpac.h>

void handler_proc(long signal, long value, CxMechanism mech)
{
    printf("\nsignal 10 has a new value of %d\n", value);
    CxSendMechanismErrorAction ( mech, CX_MECH_ABORT ) ;

    .
    .
    .
}

void main (void)
{
    CxServer Server ;
    CxMechanism mech ;
    .
    .
    /* Open Server and mechanism */
    .
    .
```
/* Set up interrupt on lower thresh */
    CxInterruptOnLowerThreshold(Server, 10, 5, mech, handler_proc);
    
    /* Sleep until interrupt is awakened. */
    while(1)
        pause();
    
}

SEE ALSO
    CxInterruptOnLowerThresholdOff, CxSendMechanismErrorAction,
    CxInterruptOnUpperThreshold
## CxInterruptOnLowerThresholdOff

Disables a signal handler for lower threshold monitoring

### SYNOPSIS

```c
#include <code/robpac.h>
long CxInterruptOnLowerThresholdOff (CxServer Server, long signal_num)
```

### ARGUMENTS

- **Server**
  - Server ID
- **signal_num**
  - Signal number

### DESCRIPTION

This function disables a signal interrupt handler, previously registered using the `CxInterruptOnLowerThreshold` function. After this function is called, the interrupt handler previously registered will no longer be invoked if the signal goes below the specified threshold, unless `CxInterruptOnLowerThreshold` is called again.

Any mechanism specified in the corresponding `CxInterruptOnChange` function will no longer be halted if the signal changes state, after this function is executed.

This function will return an error if the threshold has already been passed and the interrupt has not been re-registered before the `CxInterruptOnLowerThresholdOff` is called.

### RETURN VALUES

This function returns **0** if successful; otherwise, **-1 (CX_ERROR)** is returned.

### ERRORS

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_NO_SUCH_ENTRY</td>
<td>A <code>CxInterruptOnLowerThreshold</code> handler has not been registered with the CIMServer for the given signal, or the threshold has been passed and the interrupt not re-registered.</td>
</tr>
</tbody>
</table>

### EXAMPLE

The following program first sets a signal value to 100, and registers interrupt handler for a signal value going below a specified limit. Next, it changes the value of the signal. Whenever a signal value goes below the specified limit, the interrupt handler is called and a message is displayed. In this case, the interrupt handler resets the signal value then re-enables itself. (The latter step is necessary because interrupt handlers are automatically disabled after each call). The interrupt function resets the signal value and re-enables itself. The program then disables the interrupt handler by making calling `CxInterruptOnThresholdOff`. After that, changes to the signal value will no longer trigger the interrupt handler.

```c
#include <stdio.h>
```
#include <code/robpac.h>
#include "sigTable.h"          /* define signals and their states */

void value_below_threshold(long signal, long value, CxMechanism mech);

void main (void)
{
    CxServer Server;
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    CxSetSignal(Server, SW1_SIGNAL, 100);
    CxInterruptOnLowerThreshold( Server, SW1_SIGNAL, 10, CX_NULL, value_below_threshold );
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxSetSignal( Server, SW1_SIGNAL, 5 );
    CxSetSignal( Server, SW1_SIGNAL, 0 );
    CxInterruptOnLowerThresholdOff(Server, SW1_SIGNAL);
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxSetSignal( Server, SW1_SIGNAL, 5 );
    CxRobpacExit ();
}

void value_below_threshold(long signal, long value, CxMechanism mech)
{
    printf(" Signal value reached below threshold. Value : %ld \n", value );
    /* must reset signal value to avoid infinite interrupt */
    CxSetSignal(Server, SW1_SIGNAL, 100);
    CxInterruptOnLowerThreshold( Server, SW1_SIGNAL, 10, CX_NULL, value_below_threshold );
}

SEE ALSO
CxInterruptOnLowerThreshold, CxInterruptOnUpperThresholdOff, CxInterruptOnUpperThreshold
**CxInterruptOnUpperThreshold**

Sets up a signal handler for upper threshold monitoring

**SYNOPSIS**

```
#include <code/robpac.h>
long CxInterruptOnUpperThreshold(CxServer Server, long signal_num, long value, CxMechanism mech, void (*handler_proc)())
```

**ARGUMENTS**

- **Server**
  Server ID
- **signal_num**
  Signal number
- **value**
  Signal value above which an interrupt handler will be called
- **mech**
  Mechanism ID of mechanism open for control. The mechanism will stop when the signal rises above or becomes equal to the specified threshold. If no mechanism is included, a CX_NULL pointer may be used.
- **handler_proc**
  User-supplied signal handler

**DESCRIPTION**

This function causes a user-supplied signal handling routine to be invoked whenever the value of the specified signal rises above or equal to the specified value. Upon return from the signal handler, the process resumes from the pre-interrupt location in the application.

The defined signal can be either an input or an output signal. A mechanism can be halted when the specified signal reaches its upper threshold state. This is done by passing a valid mechanism pointer, obtained using the `CxOpenMechanism` function, through the `mech` argument.

Once the signal is triggered and the mechanism is halted, the process must specify the default action to be taken on the current motion, and any other motions pending in the move queue on the CIMServer. This is done using the `CxSendMechanismErrorAction` function. The possible actions, which can be taken when this function is invoked, are the following:

- Abort the current motion and flush the remaining moves in the queue;
- Ignore the rest of the current move and process the remaining moves in the queue, or
- Resume the current move and finish processing the remaining moves in the queue

**NOTE:** If a subsequent call is made to `CxInterruptOnUpperThreshold` without disabling the previous call and with the same signal number and value, the previous mechanism and original handler function are overwritten by the new mechanism and signal handler function.

The user’s handler is invoked when the value of the signal rises above the specified threshold. The handler stays active until it is disabled by a call to the `CxInterruptOnUpperThresholdOff` function. `CxInterruptOnUpperThreshold` is automatically disabled after the threshold is passed once. To use this function more than once, it should be reenabled in the interrupt handler routine.

If the signal is still in the specified range when this function is reset, the interrupt handler gets called immediately. To avoid possible infinite loops, wait for the signal to fall below the threshold level before resetting `CxInterruptOnUpperThreshold`.

The interrupt routine should have the following format, and functional arguments:

```
void handler_proc ( long signal, long value, CxMechanism mech )
```

The arguments for this routine are the following:
The signal which triggered the interrupt

The current state of the signal

Mechanism ID which was registered with the interrupt handler. The mechanism must be open for CX_CONTROL. CX_NULL means that no interaction with the mechanism is desired.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_NO_SIGNAL_FUNC_ROUTINE</td>
<td>The driver type specified in the signal table has not been implemented in the cntrl_signal_func function in the CIMServer.</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
<tr>
<td>CX_INVALID_MECHANISM</td>
<td>An invalid mechanism pointer was passed in through the mech argument.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

This example causes the user function handler_proc to be invoked whenever signal 10 rises above the value of 5. When this occurs, the mechanism is halted and all motions pending in the queue are aborted. The program simply sleeps at all other times.

```c
#include <stdio.h>
#include <code/robpac.h>

void handler_proc(long signal, long value, CxMechanism mech)
{
    printf("signal 10 has a new value of %ld\n", value);
    CxSendMechanismErrorAction ( mech, CX_MECH_ABORT ) ;
}

void main (void)
{
    CxServer Server ;
    CxMechanism mech ;
    
    /* Open Server and mechanism */
```
/* Set up interrupt on upper threshold */
CxInterruptOnUpperThreshold(Server, 10, 5, mech, handler_proc);

/* Sleep until interrupt is awakened. */
while(1)
   pause();

SEE ALSO
   CxInterruptOnUpperThresholdOff, CxSendMechanismErrorAction,
   CxInterruptOnValue
**CxInterruptOnUpperThresholdOff**

Disables a signal handler for upper threshold monitoring

**SYNOPSIS**
```c
#include <code/robpac.h>
long CxInterruptOnUpperThresholdOff(CxServer Server, long signal_num)
```

**ARGUMENTS**
- **Server**: Server ID
- **signal_num**: Signal number

**DESCRIPTION**
This function disables a signal interrupt handler, previously registered using the `CxInterruptOnUpperThreshold` function. Once called, the interrupt handler previously registered will no longer be invoked if the signal reaches the specified value.

Any mechanism specified in the corresponding `CxInterruptOnChange` function will no longer be halted if the signal changes state, after this function is executed.

This function will return an error if the threshold has already been passed and the interrupt has not been re-registered before the `CxInterruptOnLowerThresholdOff` is called.

**RETURN VALUES**
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_NO_SUCH_ENTRY</td>
<td>A <code>CxInterruptOnUpperThreshold</code> handler has not been registered with the CIMServer for the given signal, or the threshold has been passed and the interrupt not reset.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
The following program sets up a signal value to 0.0 and an interrupt handler for a signal value going above a specified limit. Next, it changes the value of the signal. Whenever a signal value goes above the specified limit the first time, the interrupt handler is called and a message is displayed. The interrupt handler is automatically disabled. The function resets the signal value and re-enables itself. The program then turns off the interrupt handler by making a specific call to `CxInterruptOnUpperThreshold`. After that, changing the signal value above the specified limit does not trigger the interrupt handler.

```c
#include <stdio.h>
#include <code/robpac.h>
```
#include "sigTable.h"          /* define signals and their states */

void value_above_threshold(long signal, long value, CxMechanism mech);
CxServer    Server;

void main(void)
{
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    CxSetSignal(Server, SW1_SIGNAL, 0);
    CxInterruptOnUpperThreshold( Server, SW1_SIGNAL, 90, CX_NULL,
                                  value_above_threshold );
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxSetSignal( Server, SW1_SIGNAL, 90 );
    CxSetSignal( Server, SW1_SIGNAL, 95 );
    CxInterruptOnUpperThresholdOff(Server, SW1_SIGNAL);
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxSetSignal( Server, SW1_SIGNAL, 90 );
    CxRobpacExit ();

    void value_above_threshold(long signal, long value, CxMechanism mech)
    {
        printf(" Signal value reached above threshold. Value : %ld \n", value )
        /* must reset signal value to avoid infinite interrupt */
        CxSetSignal(Server, SW1_SIGNAL, 0 );
        CxInterruptOnUpperThreshold( Server, SW1_SIGNAL, 90, CX_NULL,
                                      value_above_threshold );
    }

SEE ALSO
    CxInterruptOnUpperThreshold, CxInterruptOnLowerThreshold
**CxInterruptOnValue**

Sets up a signal handler for a specified signal value

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxInterruptOnValue(CxServer Server, long signal_num, long value,
                        CxMechanism mech, void (*handler_proc)())
```

**ARGUMENTS**

- **Server**
  - Server ID
- **signal_num**
  - Signal number
- **value**
  - Specific value that causes interrupt
- **mech**
  - Mechanism ID. The mechanism, which must be open for CX_CONTROL, will stop when the signal reaches the specified value. If no mechanism is specified, a CX_NULL pointer may be used.
- **handler_proc**
  - User supplied signal handler

**DESCRIPTION**

This function causes a user-supplied signal handling routine to be invoked whenever the signal in question reaches the specified state. Upon return from the signal handler, the process resumes from the pre-interrupt location in the application. The defined signal can be either an input or an output signal.

This function can also be used to halt a mechanism when the signal reaches the specified value. This is done by passing a mechanism pointer returned by `CxOpenMechanism`.

A mechanism can be halted when the specified signal reaches the specified state. This is done by passing a valid mechanism pointer returned from the `CxOpenMechanism` function.

Once the signal is triggered and the mechanism is halted, the process must specify the default action to be taken on the current motion, and any other motions pending in the move queue on the CIMServer. This is done using the `CxSendMechanismErrorAction` function. The possible actions, which can be taken when this function is invoked, are the following:

- Abort the current motion and flush the remaining moves in the queue;
- Ignore the rest of the current move and process the remaining moves in the queue, or
- Resume the current move and finish processing the remaining moves in the queue;

**NOTE:** If a subsequent call is made to `CxInterruptOnValue` with the same signal, the previous mechanism and signal handler functions are overwritten by the new values.

The user handler stays active until it is disabled a call to the `CxInterruptOnValueOff` API function. The interrupt routine should have the following format, and functional arguments:

```c
void handler_proc ( long signal, long value, CxMechanism mech )
```

The arguments for this routine are the following:
**Signal**  The signal which triggered the interrupt
**Value**  The current state of the signal
**Mech**  Mechanism ID which was registered with the interrupt handler. Mechanism must be open for **CX_CONTROL**, **CX_NULL** if no interaction with the mechanism is desired.

**RETURN VALUES**
This function returns **0** if successful; otherwise, **-1** (**CX_ERROR**) is returned.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the **CxGetErrorNumber** function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CX_MESSAGE_SEND_FAILED</strong></td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td><strong>CX_MESSAGE_RECEIVE_FAILED</strong></td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td><strong>CX_NON_EXISTENT_SIGNAL</strong></td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td><strong>CX_NO_SIGNAL_FUNC_ROUTINE</strong></td>
<td>The driver type specified in the signal table has not been implemented in the <strong>cntrl_signal_funct</strong> function in the CIMServer.</td>
</tr>
<tr>
<td><strong>CX_MECH_NOT_OPEN_FOR_CONTROL</strong></td>
<td>Mechanism is not open for control.</td>
</tr>
<tr>
<td><strong>CX_INVALID_MECHANISM</strong></td>
<td>An invalid mechanism pointer was passed in through the mech argument.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
This example causes the user function **light_curtain** to be invoked whenever the signal **LIGHT_CURTAIN** changes to **BROKEN**. If the signal is activated, the mechanism’s motion is halted, and all pending moves are aborted. The program simply sleeps at all other times.

```c
#include <stdio.h>
#include <code/robpac.h>
#include "mysignals.h"

#define BROKEN 1

void light_curtain( long signal, long value, CxMechanism mech);

void main (void)
{
    CxServer Server ;
    CxMechanism mech ;

    /* Open server and mechanism */
    
    /* Set up interrupt, then sleep till signal is triggered. */
    CxInterruptOnValue(Server, LIGHT_CURTAIN, BROKEN, mech,
```
light_curtain);
    
    while(1)
        pause();
    }

void light_curtain( long signal, long value, CxMechanism mech)
{
    printf("WARNING--someone broke the light curtain\n");
    CxSendMechanismErrorAction ( mech, CX_MECH_ABORT ) ;
}

SEE ALSO
    CxInterruptOnValueOff, CxSendMechanismErrorAction,
    CxInterruptOnLowerThreshold, CxInterruptOnUpperThreshold
CxInterruptOnValueOff

Disables a signal handler for a signal change to a specific value

SYNOPSIS

#include <code/robpac.h>
long CxInterruptOnValueOff(CxServer Server, long signal_num, long value)

ARGUMENTS

Server Server ID
signal_num Signal number
value Value that the signal must change to, as specified in the CxInterruptOnValue function

DESCRIPTION

This function disables a signal interrupt handler, previously registered using the CxInterruptOnValue function. Once called, the interrupt handler previously registered will no longer be invoked if the signal reaches the specified value. Similarly, if a mechanism was specified in the corresponding CxInterruptOnValue function, then after this function is executed the mechanism will no longer be halted if the signal reaches the specified state.

The call to CxInterruptOnValueOff will only disable the signal handler specified by the CxInterruptOnValue function made with the same signal_num and value arguments. All other CxInterruptOnValue calls made for the same signal, but different values remain active.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer's signal table.</td>
</tr>
<tr>
<td>CX_NO_SUCH_ENTRY</td>
<td>A CxInterruptOnValue handler has not been registered with the CIMServer for the given signal and value.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program sets a signal value to 0 and registers an interrupt handler for a specific signal value. Next, it changes the value of the signal. When the signal value equals the specified value, the interrupt handler is called and a message is displayed. The handler is then disabled, after which it is no longer involved when the signal changes to the previously specified value.
#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h" /* define signals and their states */

void value_reached(long signal, long value, CxMechanism mech);

CxServer Server;

void main(void)
{
    Server = CxOpenServer( "Testing", CX_SMEM, 0 );

    CxSetSignal(Server, SW1_SIGNAL, 0 );
    CxInterruptOnValue(Server, SW1_SIGNAL, 50, CX_NULL, value_reached);
    CxSetSignal( Server, SW1_SIGNAL, 10 );
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxInterruptOnValueOff( Server, SW1_SIGNAL, 50 );
    CxSetSignal( Server, SW1_SIGNAL, 10 );
    CxSetSignal( Server, SW1_SIGNAL, 50 );
    CxRobpacExit ();
}

void value_reached(long signal, long value, CxMechanism mech)
{
    printf(" Signal value reached setting. Value : %ld \n", value);
}

SEE ALSO
    CxInterruptOnValue
CxPositionComp
Sets a signal value when an axis position is reached.

SYNOPSIS
#include <code/robpac.h>
long CxPositionComp(CxMechanism mech, long axis, long num_pos, double *comp_pos, long status_sig, long output_sig, long output_value, double width, long mode, double rollover)

ARGUMENTS
mech               mechanism ID
axis               The number of the axis in the mech for position comparing
num_pos            The number of positions being compared. This must not exceed CX_MAX_POS_COMP.
comp_pos           An array of size num_pos containing compared position values
status_sig         Signal used for counting the number of comparisons made. This allows an application to wait for a specific comparison to be made. The allowable signals are hardware-dependent (see Hardware & System Dependencies below).
output_sig         An output signal which is set to output_value when a successful comparison occurs. For some servo cards, this functionality is implemented in hardware, making it very fast and accurate (see Hardware & System Dependencies below).
output_value       The value to which output_sig is set. Allowable values may be hardware dependent (see Hardware & System Dependencies below).
width              The distance or time period over which the output signal stays at output_value before being restored to its original state (see Hardware & System Dependencies below).
mode               A flag which can assume one of three values: CX_ONE_SHOT, CX_ROLL_OVER and CX_FIFO.
rollover           The rollover value to be used when mode=CX_ROLL_OVER.

DESCRIPTION
CxPositionComp allows a signal to be set at any position along a move, and is typically used to strobe a line scan camera. If output_sig is a hardware output on the servo card, this operation is typically performed in hardware, so the output is set almost instantaneously (within a few nanoseconds). The status signal is incremented as the axis position reaches the corresponding positions in a user-specified array. The integer values start from zero, incrementing by one when each successive position in the array is reached. Applications typically use the status signal to monitor positions and use the output signal for rapid response when no host interaction is required.

At present, this function is only supported for use with the PMAC and MEI DSP servo cards. There are some differences in the behavior of this function on these cards, due to differences in their capabilities (see Hardware & System Dependencies for further details). The mode parameter specifies the manner in which CIMControl is to iterate through the list of positions in comp_pos.
• If mode equals CX_ONE_SHOT, CIMControl triggers on each position only once, and in sequence (i.e. starting with comp_pos[0] and ending with comp_pos[num_pos-1]).

• If mode is set to CX_ROLL_OVER, then CIMControl triggers on each position in sequence, adds the rollover value to each position, then repeats iterating through comp_pos until CxDisablePosComp is called.

• If mode equals CX_FIFO, CIMControl triggers on each position only once and sequentially, just as when mode is set to CX_ONE_SHOT. However, this option allows additional positions to be dynamically loaded by calling CxAddToPosComp.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns -1(CX_ERROR), the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMControl.</td>
</tr>
<tr>
<td>CX_MESSAGE.Receive_FAILED</td>
<td>Error in receiving message from the CIMControl.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The status or output signal is not defined in the signal table.</td>
</tr>
<tr>
<td>CX_SIGNAL_NOT_VALID</td>
<td>The status or output signal is not a valid choice for position compare (see Hardware &amp; System Dependencies below).</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The input values are invalid arguments in the function definition.</td>
</tr>
</tbody>
</table>

EXAMPLES

```
#include <code/robpac.h>
#include "mySignals.h"

void main( void )
{
    double compPos[CX_MAX_POS_COMP];
    double rollover;
    CxServer Server;
    CxMechanism Mech;
    ....
    
    /* Open CxServer and CxMechanism */
    ....
    
    rollover = 360.0;
    compPos[0] = 60.0;
    compPos[1] = 120.0;
    compPos[2] = 180.0;
```
compPos[3] = 240.0;
CxMoveSingleAxis( Mech, 0, -.01);
CxPositionComp( Mech, 0, 4, compPos, COMPARE_STATUS, STROBE_CAMERA, 1, 5.0, 1, rollover );
CxMoveSingleAxis( Mech, 0, 3600.0 );
/* 4 compares per revolution * 10 revolutions = 40 hits */
for(i==0; i<40; i++) {
    CxWaitForChange(Server, COMPARE_STATUS, &value, 0.0);
    printf("At position %/f\n", compPos[value]);
}

HARDWARE & SYSTEM DEPENDENCIES
Use of status_sig is only supported on the PMAC motion card. On the MEI DSP card, this signal will not be incremented; nevertheless, it must be a valid signal since it is cleared by CIMControl when CxPositionComp is invoked. It should therefore be assigned to an unused software or MEI DSP signal.

On the PMAC card, the width parameter signifies the distance over which the output signal is pulsed before returning to its original value. On the MEI DSP card, this is the time interval in milliseconds over which the signal is pulsed.

Finally, only the CX_ONE_SHOT mode is supported for use on the MEI DSP card. In contrast, all three modes (CX_ONE_SHOT, CX_ROLL_OVER and CX_FIFO) are supported for use with the PMAC card.

SEE ALSO
CxDisablePosComp
CxPulseSignal
Changes a signal’s value for a specified length of time, then resets the signal value

SYNOPSIS
#include <code/robpac.h>
long CxPulseSignal(CxServer Server, long signal_num, long value, long time)

ARGUMENTS
Server Server ID.
signal_num Signal number
value New value for the signal.
time Time in milliseconds to pulse the signal

DESCRIPTION
This function sets a signal’s value for a specified amount of time, and then restores it to the state it had before 
CxPulseSignal was called.
If the CIMServer is in CX_RUNTIME mode, the signal must be an output signal; otherwise, an error will 
occur. If it is in CX_ANIMATION mode, any signal can be pulsed.
This function does not return until after the signal has been pulsed. In order to pulse a signal, then return 
without waiting for the pulse to complete, the CxAsyncPulseSignal function should be used.
It is possible for another function to change the signal value before the pulse is completed; however, the 
signal restored to the value it had prior to calling CxPulseSignal.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber 
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NONEXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The specified signal is not an output signal, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

EXAMPLE
The following program sets up a connection with the server. It reads the value of a signal and enters a loop to continuously monitor the signal value. Since the signal was pulsed synchronously, the program does not resume executing until the pulse is over. When the program does resume execution, the pulse signal value is always high, the program never enters the while loop and a message is never printed out.
The program then waits one second and starts a five second asynchronous pulse. Afterwards, control will come back to the program, which will poll the signal status until the pulse is over.

**NOTE:** When the server is in **CX_ANIMATION** mode, **CxAsyncPulseSignal** behaves the same as **CxPulseSignal**.

**NOTE:** **PULSE_SIGNAL**, **PULSE_HIGH**, and **PULSE_LOW** are assumed to be signal table indexes defined in **sigTable.h**.

```c
#include <stdio.h>
#include <code/robpac.h>
#include "sigTable.h"
#define PULSE_HIGH 1
#define PULSE_LOW 0

void main ( void )
{
  CxServer Server;
  long pulse_state;

  Server = CxOpenServer( "Testing", CX_SMEM, 0 );

  /* Start and wait till the first pulse is over */
  CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
  CxPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
  CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
  while (pulse_state != PULSE_HIGH)
  {
    /* This portion will never get executed */
    printf(" Pulse is still not over \n");
    CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
  }
  printf(" The first pulse signal is over \n");

  CxDelay(1,0);

  /* Start and wait till the second pulse is over */
  CxSetSignal(Server, PULSE_SIGNAL, PULSE_HIGH);
  CxAsyncPulseSignal(Server, PULSE_SIGNAL, PULSE_LOW, 5000);
  CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
  while (pulse_state != PULSE_HIGH)
  {
    printf(" Pulse is still not over \n");
    CxGetSignalValue(Server, PULSE_SIGNAL, &pulse_state);
  }
  printf(" The second pulse signal is over \n");

  CxRobpacExit();
}

SEE ALSO

CxAsyncPulseSignal, CxSetSignal, CxGetSignalValue
CxQueuedSetSignal

Sets the value of a signal through the motion queue

SYNOPSIS

#include <code/robpac.h>
long CxQueuedSetSignal(CxMechanism mech, long signal_num, long value)

ARGUMENTS

mech   Mechanism ID
signal_num   Signal number
value   New value for the signal

DESCRIPTION

This function sets the state or value of a given signal. The signal is set only after any moves and signals which
are ahead of this signal in the motion queue have completed. Thus, this function can be used to coordinate the
setting of an output with the completion of one or more queued moves.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber
function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE.Receive_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The signal is not defined as an output, and the Server is in CX_RUNTIME mode.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following code fragment defines a reusable function for placing objects. The user specifies an object to
be placed, a location for placing the object (defined as a node frame), a tool to use, an approach and retraction
distance, and a signal and value for actuating the gripper.

    .
    .
place_object (CxServer Server, CxMechanism mech, CxNodeId location,
          CxNodeId tool, double z_dist, long signal, long value)
{    
    CxMoveNearNode(mech, location, tool, z_dist);
    CxMoveToNode(mech, location, tool);
    CxQueuedSetSignal(mech, signal, value);
    CxMoveAway(mech, tool, z_dist);
}
The function might then be called as follows:

place_object(Server, mech, loc, gripper, 100.0, OPEN_HAND, CX_TRUE);

HARDWARE AND SYSTEM DEPENDENCIES

When used with a PMAC motion card, CxQueuedSetSignal does not take effect immediately. Rather, it must be followed by either a move command (e.g. CxMoveAllAxes or CxWaitForEndOfMotion). CxQueuedSetSignal will not be activated until this command has started executing. Similarly, if a motion error should cause the move command or CxWaitForEndOfMotion to abort, CxQueuedSetSignal will be aborted as well.

SEE ALSO

CxQueuedWaitForEvent
**CxQueuedWaitForSignal**

Wait for the value of a signal through the motion queue

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxQueuedWaitForSignal(CxMechanism mech, long signal_num, long value, long type, long timeout)
```

**ARGUMENTS**

- `mech`: mechanism ID
- `signal_num`: Signal number
- `value`: New value for the signal
- `type`: `CX_SPECIFIC_VALUE_CHANGE` or `CX_GREATER_THAN_CHANGE` or `CX_LESS_THAN_CHANGE`
- `timeout`: Timeout value in milliseconds

**DESCRIPTION**

This function waits for the value of a given signal. The API function requires that the controller type for the signal and for the mechanism be the same. The API function waits for any moves and signals which are ahead of this signal in the motion queue to be completed. Then it waits for the signal to have the specified value.

**RETURN VALUES**

This function returns 0 if successful; otherwise, `-1(CX_ERROR)` is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CX_MESSAGE_SEND_FAILED</code></td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td><code>CX_MESSAGE_RECEIVE_FAILED</code></td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td><code>CX_NON_EXISTENT_SIGNAL</code></td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td><code>CX_QUEUE_IN_BLOCK_STATUS</code></td>
<td>The motion queue is in block status, and <code>CxSendMechanismErrorAction()</code> for <code>CX_MECH_ABORT</code> has to be called to clear this status.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following code fragment defines a reusable function for placing objects. The user specifies an object to be placed, a location for placing the object (defined as a node frame), a tool to use, an approach and retraction distance, and a signal wait for value condition to be true.

```c
place_object (CxServer Server, CxMechanism mech, CxNodeId location, CxNodeId tool, double z_dist, long signal, long value)
```
The function might then be called as follows:

```c
place_object(Server, mech, loc, gripper, 100.0, OPEN_HAND, CX_TRUE);
```

SEE ALSO

CxQueuedSetSignal
CxSetScanRate
Changes the polled signal scan rate

SYNOPSIS
#include <code/robpac.h>
long CxSetScanRate(CxServer Server, long scan_rate);

ARGUMENTS
Server Server ID
scan_rate Rate in milliseconds

DESCRIPTION
If CIMControl is in CX_RUNTIME mode and the signal table contains hardware input signals which must be polled, the CxSetScanRate function sets the rate at which these input signals are polled.
By default the scan rate is 20 ms.
The specified scan rate must be greater than 10 ms; otherwise, an error will occur.

RETURN VALUES
This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving a message from the CIMServer.</td>
</tr>
<tr>
<td>CX_SCAN_RATE_TOO_SMALL</td>
<td>The given scan rate is smaller than min_scan_rate (10 ms).</td>
</tr>
</tbody>
</table>
**CxSetSignal**

Sets the value of a signal

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSetSignal(CxServer Server, long signal_num, long value)
```

**ARGUMENTS**

- **Server**  
  Server ID
- **signal_num**  
  Signal number
- **value**  
  New value for the signal

**DESCRIPTION**

This function sets the state or value of a given signal. If the CIMServer is in `CX_RUNTIME` mode and the signal is not a `CX_SOFTWARE` signal, the signal must have its output attribute turned on in the signal table. If the server is not in `CX_RUNTIME` mode, the value of any signal can be set.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (`CX_ERROR`) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CX_MESSAGE_SEND_FAILED</code></td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td><code>CX_MESSAGE_RECEIVE_FAILED</code></td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td><code>CX_NON_EXISTENT_SIGNAL</code></td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td><code>CX_CANT_SET_INPUT_SIGNAL</code></td>
<td>The signal is not defined as an output, and the Server is in <code>CX_RUNTIME</code> mode.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following code fragment defines a reusable function for placing objects. The user specifies an object to be placed, a location for placing the object (defined as a node frame), a tool to use, an approach and retraction distance, and a signal and value for actuating the gripper.

```c
place_object (CxServer Server, CxMechanism mech, CxNodeId object, CxNodeId location, CxNodeId tool, double z_dist, long signal, long value)
{
    CxMoveNearNode(mech, location, tool, z_dist);
    CxMoveToNode(mech, location, tool);
    CxSetSignal(Server, signal, value);
    CxAttachNode(Server, object, location);
}
```
```
CxMoveAway(mech, tool, z_dist);
}

The function might then be called as follows:

place_object(Server, mech, Peg, Hole, gripper, 100.0,
             OPEN_HAND, CX_TRUE);

SEE ALSO

CxGetSignalValue, CxPulseSignal
```
**CxWaitForChange**

Waits for a specific signal to change its current state

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxWaitForChange(CxServer Server, long signal_num, long *value, long timeout)
```

**ARGUMENTS**

- `Server`: Server ID
- `signal_num`: Signal number
- `value`: Returns the current value of the signal
- `timeout`: Maximum time in milliseconds to wait for the condition to be met; a 0 argument makes this time infinite.

**DESCRIPTION**

This function causes a process to sleep until either the timeout period has expired or the specified signal changes state. The timeout period can be made infinite (no timeout) by setting the timeout argument to zero. The return value indicates if a state change occurred or if the function timed out.

The specified signal can be either an input or an output signal. If the condition does not time out, the current value of the signal is passed back to the user through the value parameter.

**RETURN VALUES**

This function returns `CX_TIMED_OUT` if time expires, `-1` (CX_ERROR) on error, or `0` if successful.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function.

The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

This code in this example waits one second for signal 10 to change state.

```c
long ret_code;
long value;
CxServer Server ;
```
/* Open server, etc */
.
.
/* Wait for signal 10 to change value */
ret_code=CxWaitForChange(Server, 10, &value, 1000);
if(ret_code != CX_ERROR)
{
    if(ret_code == CX_TIMED_OUT)
        printf("no state changed occurred\n");
    else
        printf("signal 10 changed state %d\n", value);
}
.
.
SEE ALSO

CxWaitForValue, CxWaitForUpperThreshold, CxWaitForLowerThreshold
**CxWaitForLowerThreshold**
Waits for a signal to change to any value below a given threshold

**SYNOPSIS**
```
#include <code/robpac.h>
long CxWaitForLowerThreshold(CxServer Server, long signal_num,
   long *value, long timeout)
```

**ARGUMENTS**
- **Server**: Server ID
- **signal_num**: Signal number
- **value**: Variable used to set the threshold value and return the actual value of the signal when the threshold is exceeded
- **timeout**: Maximum time in milliseconds to wait for the condition to be met; a 0 argument makes this time infinite

**DESCRIPTION**
This function causes a process to sleep until either the timeout period has expired or the specified signal reaches or drops below the specified threshold. The timeout period can be made infinite (no timeout) by setting the timeout argument to zero. The return value indicates if the condition was met or if the function timed out.

The specified signal can be either an input or an output signal. If the wait condition does not time out, the current value of the signal is passed back through the value parameter.

This function is generally used with analog signals.

**RETURN VALUES**
This function returns `CX_TIMED_OUT` if time expires, `-1` (CX_ERROR) if an error occurs, or `0` if successful.

**ERRORS**
If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CX_MESSAGE_SEND_FAILED</code></td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td><code>CX_MESSAGE_RECEIVE_FAILED</code></td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td><code>CX_NON_EXISTENT_SIGNAL</code></td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

**EXAMPLES**
This example waits one second for a signal indicating a touch probe is no longer in contact:
```
#include <code/robpac.h>
#include "sigTable.h" /* signal table constant definitions */
#define NOT_TOUCHING20
```
long ret_code, value;
CxServer Server;

/*/ Open server, etc */

/*/ Wait for lower threshold */
value = NOT_TOUCHING;
ret_code=CxWaitForLowerThreshold(Server, TOUCH_PROBE,
              &value, 1000);
if(ret_code != CX_ERROR)
{
    if (ret_code == CX_TIMED_OUT
        printf("still touching\n");
    else
        printf("not touching.. value = %ld\n", value);
}

SEE ALSO
CxWaitForChange, CxWaitForUpperThreshold, CxWaitForValue
CxWaitForUpperThreshold
Waits for a signal to change to any value above a given threshold

SYNOPSIS
#include <code/robpac.h>
long CxWaitForUpperThreshold(CxServer Server, long signal_num, long *value, long timeout)

ARGUMENTS
Server Server ID
signal_number Signal number
value Variable used to set the threshold value and return the actual value of the signal when the threshold is exceeded
Timeout Maximum time in milliseconds to wait for the condition to be met; a 0 argument makes this time infinite

DESCRIPTION
This function causes a process to sleep until either the timeout period has expired or the signal reaches or rises above the specified threshold. The timeout period can be made infinite (no timeout) by setting the timeout argument to zero. The return value indicates whether the condition was met or if the function timed out.
The specified signal can be either an input or an output signal. If the wait condition does not time out, the current value of the signal is passed back through the value parameter.
This function is generally used with analog signals.

RETURN VALUES
This function returns CX_TIMED_OUT if time expires, -1 (CX_ERROR) if an error occurs, or 0 if successful.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

EXAMPLES
This example waits 1 second for a signal indicating a touch probe has made contact.

#include <code/robpac.h>
#include "sigTable.h" /* signal table constant definitions */
#define NOT_TOUCHING20
long ret_code, value;
CxServer Server;

/* Open server, etc */

/* Wait for lower thresh */
value = NOT_TOUCHING;
ret_code=CxWaitForUpperThreshold(Server, TOUCH_PROBE, &value, 1000);
if(ret_code != CX_ERROR)
{
    if(ret_code == CX_TIMED_OUT)
        printf("never touched\n");
    else
        printf("touched with value = %ld\n", value);
}

SEE ALSO

CxWaitForChange, CxWaitForLowerThreshold, CxWaitForValue
CxWaitForValue
Waits for a signal to change to a specific value

SYNOPSIS
#include <code/robpac.h>
long CxWaitForValue(CxServer Server, long signal_num, long value, long timeout)

ARGUMENTS
Server Server ID
signal_num Signal number
value Signal value that will terminate the wait
timeout Maximum time in milliseconds to wait for the condition to be met; a 0 argument makes this time infinite

DESCRIPTION
This function causes a process to sleep until either the timeout period has expired or the specified signal state or value has been reached. The timeout period can be made infinite (no timeout) by setting the timeout argument to zero. The return value indicates whether the condition was met or if the function timed out.
The specified signal can be either an input or output.

RETURN VALUES
This function returns CX_TIMED_OUT if time expires, -1 (CX_ERROR) if an error occurs, or 0 if successful.

ERRORS
If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
</tbody>
</table>

EXAMPLES
This example waits one second for a signal indicating the part has been gripped.
#include <code/robpac.h>
#include "sigTable.h" /* signal table constant definitions */
#define PART_GRIpped 1

long ret_code;
CxServer Server;
/* Open Server, etc */

/* Wait for GRIPPER signal to have value PART_GRIPPED */

ret_code=CxWaitForValue(Server, GRIPPER, PART_GRIPPED, 1000);
if(ret_code != CX_ERROR)
{
    if(ret_code == CX_TIMED_OUT)
        printf("part gripping failed\n");
    else
        printf("part was gripped\n");
}

SEE ALSO

CxWaitForChange,CxWaitForUpperThreshold,CxWaitForLowerThreshold
**CxWhen**

Sets the value of one signal when another signal changes to a specified state

**SYNOPSIS**

```
#include <code/robpac.h>
CxWhen(CxServer Server, long when_signal, long when_value, long
       CxSetSignal, long set_value)
```

**ARGUMENTS**

- **Server**
  - Server ID
- **when_signal**
  - Input signal to be monitored
- **when_value**
  - Value that the input signal must change to in order to trigger the action
- **set_signal**
  - Output signal that is to be set when the conditions are met
- **set_value**
  - Value that the output signal is to be set to when the conditions are met

**DESCRIPTION**

This function allows an application process to register a signal output action to be taken when the value of another signal changes to a specified state. Once the action has been registered, the server assumes the responsibility of monitoring the specified signal and setting the specified output, thus relieving the application process of the burden of polling the signal directly.

The specified action will be executed each time the signal changes to the specified state. Once the action has been registered, it remains in effect until the application unregisters it by calling `CxDisableWhen`, or until the process exits.

If the CIMServer is in `CX_RUNTIME` mode, the `CxSetSignal` must be either a `CX_SOFTWARE` signal, or must have the output attribute turned on in the signal table. If the Server is not in `CX_RUNTIME` mode, any signal can be specified as the `CxSetSignal`.

The `CxWhen` signal can be either an input or output.

**NOTE:** To disable the `when` condition, all of the signal parameters to the function `CxDisableWhen` must be the same as those used to register the `when` condition. Otherwise an error will occur.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (`CX_ERROR`) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the `CxGetErrorNumber` function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending a message to the CIMServer.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVE_FAILED</td>
<td>Error in receiving message from the CIMServer.</td>
</tr>
<tr>
<td>CX_NON_EXISTENT_SIGNAL</td>
<td>The signal number is not defined in the CIMServer’s signal table.</td>
</tr>
<tr>
<td>CX_CANT_SET_INPUT_SIGNAL</td>
<td>The <code>set_signal</code> is not an output, and the Server is in <code>CX_RUNTIME</code> mode.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following example causes the REDLIGHT output signal to go on whenever the BADPART input signal changes to CX_TRUE. This condition is monitored until the CxDisableWhen function is called:

```c
{  CxServer myServer;
   
   /* Enable the when condition */
   CxWhen(myServer, BADPART, CX_TRUE, REDLIGHT, CX_ON);
   
   /* Disable the when condition */
   CxDisableWhen(myServer, BADPART, CX_TRUE, REDLIGHT, CX_ON);
}
```

SEE ALSO

CxDisableWhen, CxInterruptOnValue, CxWaitForValue
Exception Handling
Introduction
An overview of exception (error) handling

OVERVIEW
For any type of software development exception handling is a challenge. For machine control, this challenge is often magnified, because programmers must deal with software components and mechanical components. Liabilities for improperly handling abnormal conditions range from damaging expensive machinery and work in process to endangering human life. Complex machine control often involves several software tasks running concurrently in a multi-tasking, multi-threaded, or multi-processing environment. The actions of one task often affect the others.

The best way to deal with exceptions is to eliminate them. Careful attention to detail in the design of the physical machine and its corresponding control software can greatly reduce the number of exceptions which must be handled, resulting in a more reliable machine. It is not uncommon for the design of exception handling to take 90 percent of the time required to implement production quality machine control software. It is therefore, imperative that exception handling be designed into control software form the beginning. Saving this task for last, assuming that it can be simply added to an application, is a recipe for disaster.

This section contains a collection of tools designed to aid the programmer in implementing a comprehensive approach to exception handling. Please not that this section is meant to provide an overview of exception handling within CODE, rather than an exhaustive discussion of exception handling theory. For additional information, please consult a textbook on software engineering.

EXCEPTION TYPES
Any event which causes a program to deviate from its normal flow can be considered an exception. Exceptions are not necessarily errors; for example, an exception may be the result of sensor data alerting the application of the occurrence of an event or the result of operator intervention.

This section deals with exceptions which are either the result of calling an API function directly or are detected by the server and reported to the application. These exceptions are divided into two types: synchronous and asynchronous.

Synchronous
A synchronous exception always occurs after a direct call to one of the API functions. The exception is detected while the API function is executing and reported to the calling application when the function returns. Synchronous events are the easiest to deal with since they are always detected at a known location in the code.

Asynchronous
An asynchronous event can occur at any time and may have no correlation with the current execution point of the application. For example, there is no way to anticipate when an E-Stop button might be pressed. One of the biggest difficulties in dealing with asynchronous events is determining where the current execution point is when the exception is raised. This knowledge is vital in determining the proper course of action.

EXCEPTION HANDLING POLICIES
The system supports several exception handling models or policies for both synchronous and asynchronous events. Programmers should understand all of these policies and then choose a strategy which best fits the application requirements-and to some extent, the programmers skills. Only one policy can be in effect at any given time; however, an application can change policies at will by calling CxSetErrorPolicy.
Exit On Error

The default action is to terminate an application when any error occurs, synchronous or asynchronous. Any mechanisms being controlled by the application will be immediately halted. This is often the easiest approach—shutting the machines down and forcing the operator to manually restart the application. In many other situations though, this approach is unacceptable. The trade-off with using this policy is generally savings in programming time and programming skill requirements versus an increase in operator skill requirements and downtime.

Return Codes

The most straightforward approach to synchronous error handling is to have each API function return an error code. It is the application programmer’s responsibility to check the return code of every function and handle the error in-line. While this approach is straightforward, it requires a lot of additional programming and can make the application’s source code difficult to read.

Synchronous Exception Handler

A synchronous exception handler is essentially a filter to return codes. When an error is detected, a user-supplied function is invoked before the API function returns. If this function is capable of handling the error, it will reset the return code. This type of function is typically called a handler, and can be used in conjunction with checking return codes.

A call to CxAssignErrorHandler assigns the user function to be automatically invoked whenever an error occurs. This function must be called prior to the occurrence of any errors. Only one handler is active at any given time, so if another call to CxAssignErrorHandler is made, the new handler will replace the old one. Another useful tool is the CxSetRoblineErrorTag function, which can be used to mark certain critical sections of the application code. The current value of the tag is passed to the handler so that it can treat errors differently in different sections of the code.

Asynchronous Exception Handler With Signals

When using this method, the controller sends a signal to the application along with a message packet detailing the exception. On receipt of the signal, the application saves its context and immediately invokes the user-specified handler (this is analogous to a hardware interrupt handler). After the handler finishes processing, the original context is restored and the application continues execution from the point where the signal was detected.

As with synchronous exception handling, the CxAssignErrorHandler and CxSetRoblineErrorTag functions can be used to assign error handlers and mark critical sections of the code.

Exceptions Involving Motion

There are several ways in which motion can be abnormally terminated. An E-Stop button might be pressed, a motion error might occur, a motion-related I/O event might occur, the application might call CxStopMechanism directly, etc. For safety reasons, whenever motion is terminated abnormally, the controller requires acknowledgment from the application before any motion can be resumed. This is done by calling CxSendMechanismErrorAction. This function tells the controller how to resume processing motion commands. Whether to resume the current move, start the next move from the halted location, or flush all pending moves.

Any moves sent to the controller after motion is halted will be added to the queue of pending moves. If this queue fills, the next move sent will block. Since motion processing is halted, the queue will not drain until CxSendMechanismErrorAction is called. This can lead to potential deadlock in single threaded applications. To prevent this, exception handlers must call CxSendMechanismErrorAction before sending additional moves. Multi-threaded applications may continue to send moves to the controller as long as an exception handling thread is available to resume motion.
GETTING INFORMATION ABOUT THE CAUSE OF AN EXCEPTION

When an error occurs, all information related to that error is put into a controller specific data structure that is accessible by the application. This error structure is defined as follows:

```c
typedef struct CxErrorStruct{
    long error_id;
    long err;
    long trace_cnt
    long line [CX_ERROR_TRACE_LEN];
    long file [CX_ERROR_TRACE_LEN] [CX_MAX_ERROR_FILE+1];
    char msg[CX_ERROR_MSG_SZ];
    char errno_text[CX_MAX_ERROR_STRING];
    struct mechanism_id_msg *mech;
    long joint_num;
    long motor_num;
    long action;
    unsigned long mask;
    long l1;
    long l2;
    long l3;
    double d1;
    double d2;
    double d3;
    char time[CX_MAX_TIME];
} errorStruct;
```

The fields are defined as follows:

- **error_id**: This field holds a unique identifier for the error. This identifier is a count of errors that have occurred since the controller was last started.
- **err**: The error type identifier or number defined in roberno.h
- **trace_cnt**: The `trace_cnt` field holds the number of items in the line and file arrays.
- **line and file**: The line array contains `trace_cnt` line numbers. These are the lines of the function calls that led up to the error. The file array contains `trace_cnt` file names of files from which functions are called that led up to the error. These two fields are only useful when calling the vendor for more information about an error that occurs.
- **msg**: A detailed error message which is specific to the type of error.
- **errno_text**: A text string which is equivalent to the #defined error constant.
- **mech**: A pointer to the mechanism that realized the error. If there is no mechanism related to the error, this field will be NULL.
- **joint_num**: The number of the joint that has been the cause of error. For example, if a move was outside of joint limits, this field will contain the number of the joint that was causing the error. If the error did not involve a joint, this field will contain a -1.
- **motor_num**: This field is similar to the `joint_num` field in that it contains the number of the motor that causes the error. A joint may be driven by multiple motors, hence, the joint and motor numbers may not be identical. Again, if no motor is involved in the error, this field will contain a -1.
action

If this field is set, CxSendMechanismErrorAction needs to be called. If set to zero, this will not be necessary.

l1, l2, l3, d1, d2, d3, mask

The miscellaneous data fields l1, l2, l3, d1, d2, d3 and mask are error specific.

time

This is a time stamp of when the error occurs. The information contained in this string is day of week, month, day of month, hours, minutes, seconds, and year.

Two error structures are kept per controller. One contains asynchronous errors and the other contains synchronous errors. The reason for this is that it is possible for a synchronous and an asynchronous error to occur within close proximity to each other. Having two error structures allows the user to get both the synchronous and asynchronous information without one overwriting the other.

Macros allow the user to retrieve pointers to both error structures or to get individual members from the error structures separately. These macros can also be used to check and clear certain error conditions easily. For example, if the user needs to call CxSendMechanismErrorAction before continuing, he may check this by calling a macro. He may then call another macro to get the mechanism pointer so that he may call CxSendMechanismErrorAction without having to work interactively with his process. The following code is an example of how this would work:

```c
if( CxMoveSingleAxis( Mech, 1, 90.0 ) == CX_ERROR )
{
    CxErrorStruct * err;
    Err= CxGetError (Server);
    if( CxGetErrorAction( Server ) )
    {
        CxSendMechanismErrorAction(CxGetErrorMech(Server), CX_MECH_ABORT);
    }
}
```

The macro that returns a pointer to the current synchronous error is called CxGetError. The following code segment demonstrates how to use this macro:

```c
CxServer Server;
CxNodeId ji_id;
CxErrorStruct *my_error;
.
.
if( CxAddJoint( Server, j1_id ) == CX_ERROR){
    my_error = CxGetError( Server );
    printf( "%s[#%ld]\n",
               my_error->errno_text, my_error->err );
    printf( "%s\n", my_error->msg );
}
```

The macro that returns a pointer to the latest asynchronous error is called CxGetAsyncError. The preceding code segment works exactly the same way with both CxGetError and CxGetAsyncError. The following is a list of macros that are provided to get members of the non-asynchronous error structure:

```
CxGetErrorId, CxGetErrorNumber, CxGetErrorLine, CxGetErrorFile,
CxGetErrorMsg, CxGetErrorConstant, CxGetErrorJoint, CxGetErrorTime,
CxGetErrorMech, CxGetErrorAction, CxGetErrorMotor,
CxGetErrorTraceCount
```

Each macro takes a CxErrorStruct pointer as an argument and the return value is the type of individual element to be retrieved.
The corresponding macros for asynchronous error are called:

CxGetAsyncErrorId,  CxGetAsyncErrorNumber,  CxGetAsyncErrorLine,
CxGetAsyncErrorFile,  CxGetAsyncErrorMsg,  CxGetAsyncErrorConstant,
CxGetAsyncErrorMech,  CxGetAsyncErrorJoint,  CxGetAsyncErrorMotor,
CxGetAsyncErrorAction,  CxGetAsyncErrorTime, CxGetAsyncErrorTraceCount

Each macro takes a CxServer as an argument and the return value is the type of individual element to be retrieved.

LOGGING AND REPORTING EXCEPTIONS

The controller can be configured to log errors automatically to the screen, a file, or through a custom method provided through a user-specified function. The error logging policy is set by making a call to CxSetErrorLogPolicy. More than one error log policy can be active at any given time.

By calling CxSetErrorInfoMask, the programmer can change the amount of information reported by the error logger. This function takes a bit mask containing all of the desired information bits together. For example, to see the error number, the message and the error id, the mask would be:

CX_ERROR_NUMBER | CX_ERROR_MSG_MASK | CX_ERROR_ID.

Printing all of the error information in the default error handler, it will look something like the following:

ERROR: move_axes API failed.
CX_OUT_OF_RANGE[#47]
The requested move falls outside the limits (-45.00 to 20.00) of joint 2.
file: motion.c line: 1053
Mechanism involved: s100
Joint involved: 2
CxSendMechanismErrorAction MUST be called before proceeding
error occurred Thu Mar 30 12:50:49 1995
2 errors have occurred since CIMServer started.
**CxAssignErrorHandler**

Assigns an exception handler

**SYNOPSIS**

```c
#include <code/robpac.h>
void CxAssignErrorHandler (CxServer Server, long (*ErrorHandler)(CxServer Server, CxMechanism Mechanism, int tag, int async, char *err_buff))
```

**ARGUMENTS**

<table>
<thead>
<tr>
<th>Server</th>
<th>Server ID. This is a pointer to a CxServer returned from a previous call to CxOpenServer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorHandler</td>
<td>This is a pointer to a user-defined error handling function.</td>
</tr>
</tbody>
</table>

**ERROR HANDLER ARGUMENTS**

<table>
<thead>
<tr>
<th>Server</th>
<th>Identifies which server raised the exception. This is a pointer to a CxServer returned from a previous call to CxOpenServer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanism</td>
<td>Identifies which mechanism raised the exception. For non-motion errors, this will be a NULL pointer. This is a CxMechanism ID returned by a call to CxOpenMechanism.</td>
</tr>
<tr>
<td>tag</td>
<td>The argument is used to identify which section of the application code was active when the exception occurred. The value of tag is set whenever a call to CxSetRoblineErrorTag is made. This argument is useful if you want the exception handler to behave differently in different sections of your program. An alternative to using the tag argument is to assign different exception handlers in different sections of your program. Only one exception handler is active at any given point in time, so be sure to reset the handler each time you enter a section of your program which requires a unique handler.</td>
</tr>
<tr>
<td>async</td>
<td>If the value of this flag is 1, an asynchronous event (i.e. E-stop button pressed) occurred. If the value is 0, an asynchronous error has occurred.</td>
</tr>
<tr>
<td>Err_buff</td>
<td>This is a string containing a description of the error.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The CxAssignErrorHandler function sets the user-defined error handling function, which will be called when an error condition is reported from the CIMServer. The user handler function overrides the default error-handling function.

The error handler is invoked when the error policy is set to CX_USER_HANDLER or CX_USER_ASYNC_HANDLER (see CxSetErrorPolicy). If the error policy is set to CX_USER_HANDLER, then all error conditions will be processed through the user-defined error handler. If the error policy is set to CX_USER_ASYNC_HANDLER, then only asynchronous errors will be handled through the user-defined error handler.

**RETURN VALUES**

This function does not return any error conditions.
EXAMPLE

```
#include <stdio.h>
#include <code/robpac.h>

long MyHandler (CxServer server, CxMechanism mech, int tag, int async,
                char *err_buff);

void main (void)
{
    CxServer myServer;

    myServer = CxOpenServer ( "Server", CX_SMEM, 0 );
    if ( !myServer )
    {
        fprintf ( stderr, "CX_ERROR... CxOpenServer failed\n" );
        CxRobpacExit ( );
    }
    CxSetErrorPolicy (myServer, CX_USER_HANDLER );
    CxAssignErrorHandler(myServer, MyHandler );

    long MyHandler (CxServer server, CxMechanism mech, int tag, int async,
                    char *err_buff);
    {
        fprintf (stderr,"I got an error with code %ld\n",
                  CxGetErrorNumber(server));
        fprintf ( stderr, "%s\n", err_buff);
        return Cx_Error;
    }

HARDWARE AND SYSTEM DEPENDENCIES

Note that a valid function pointer must be passed into this function. Systems will behave unpredictably if an
invalid function pointer is passed in as the argument, and may possibly cause the process to terminate.

SEE ALSO

CxSetErrorPolicy,CxSetRoblineErrorTag
```
**CxGetAsyncError**

 Gets the structure containing information about the last asynchronous error that occurred for the server connection

**SYNOPSIS**

```c
#include <code/robpac_error.h>
CxErrorStruct *CxGetAsyncError (CxServer server)
```

**ARGUMENTS**

- `server` Connection to the current server

**DESCRIPTION**

When an asynchronous error occurs in a Cimetrix API call, all of the information about that error is stored in a CxErrorStruct structure that is a member of the server. The members of the structure are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorID</td>
<td>Number of errors that have occurred since the server connection was established</td>
</tr>
<tr>
<td>ErrorNumber</td>
<td>The Cimetrix error number associated with the error</td>
</tr>
<tr>
<td>ErrorConstant</td>
<td>The defined name of the error number</td>
</tr>
<tr>
<td>ErrorTraceCount</td>
<td>Number of items in line and file arrays</td>
</tr>
<tr>
<td>ErrorLine</td>
<td>Line number where error occurred (possible to trace more than just immediate error line)</td>
</tr>
<tr>
<td>ErrorFile</td>
<td>File where error occurred (possible to trace more than just immediate error file)</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>Text explaining the cause of the error</td>
</tr>
<tr>
<td>ErrorMechanism</td>
<td>If the error was related to a mechanism, this field contains a pointer to that mechanism</td>
</tr>
<tr>
<td>ErrorJoint</td>
<td>If the error was related to a specific joint, this field contains the joint number</td>
</tr>
<tr>
<td>ErrorMotor</td>
<td>If the error was related to a specific motor, this field contains the motor number</td>
</tr>
<tr>
<td>ErrorAction</td>
<td>Boolean flag stating whether CxSendMechanismErrorAction needs to be sent or not</td>
</tr>
<tr>
<td>ErrorTime</td>
<td>Time stamp of when the error occurred</td>
</tr>
</tbody>
</table>

Each of these fields can be accessed individually from other API calls. A pointer to this structure is returned from this call.

**RETURN VALUES**

Function returns the CxErrorStruct member of the server for asynchronous errors. If no errors have occurred, the structure will be empty.

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
                              int tag, int async, char* err_buff)
{
```

CxErrorStruct* pErrStruct;
CString strMsg, strMsg1;

if( async )
{
    pErrStruct = CxGetAsyncError( server );
    if( CxGetErrorAction( pErrStruct ) )
    {
        CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
    }

    strMsg.Format( "\tError %ld - %s: %s
At: %s
File: %s Line: ",
        CxGetAsyncErrorNumber( pErrStruct ),
        CxGetAsyncErrorConstant (pErrStruct),
        CxGetAsyncErrorMsg( pErrStruct ),
        CxGetAsyncErrorTime( pErrStruct ),
        CxGetAsyncErrorFile( pErrStruct, 0 ),
        CxGetAsyncErrorLine( pErrStruct, 0 ) );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++ )
    {
        strMsg1.Format( "\tcalled from line %ld of %s",
            CxGetAsyncErrorLine( pErrStruct, i ),
            CxGetAsyncErrorFile( pErrStruct, i ));
        strMsg += strMsg1;
    }

    strMsg1.Format( "\n%d errors have occurred since server connection",
        CxGetAsyncErrorId( pErrStruct ) );
    strMsg += strMsg1;

    MessageBox( GetDesktopWindow(), strMsg,
        MB_ICONEXCLAMATION | MB_OK |
        MB_SETFOREGROUND | MB_TOPMOST );

    return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct ) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
            CX_MECH_ABORT );
    }

    strMsg.Format( "\tError %ld - %s: %s
At: %s
File: %s Line: ",
        CxGetErrorNumber( pErrStruct ),
        CxGetErrorConstant (pErrStruct),
        CxGetErrorMsg( pErrStruct ),
        CxGetErrorTime( pErrStruct ),
        CxGetErrorFile( pErrStruct, 0 ),
        CxGetErrorLine( pErrStruct, 0 ) );
CxGetErrorCode(pErrStruct),
CxGetErrorFile(pErrStruct, 0),
CxGetErrorLine(pErrStruct, 0)
);

long lTrace = CxGetErrorTraceCount(pErrStruct);
for (int i = 1; i < lTrace; i++)
{
    strMsg1.Format("\n\tcalled from line %ld of %s",
        CxGetErrorLine(pErrStruct, i),
        CxGetErrorFile(pErrStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server connection",
    CxGetAsyncErrorId(pErrStruct));
strMsg += strMsg1;
throw(strMsg);
return 0;
}

SEE ALSO
CxGetAsynchronousError
**CxGetAsyncErrorAction**

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```
#include <codeobpac_error.h>
long CxGetAsyncErrorAction (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

PErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,  
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }
    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
 ",  
        CxGetAsyncErrorNumber( pErrStruct),
        CxGetAsyncErrorConstant (pErrStruct),
        CxGetAsyncErrorMsg( pErrStruct ),
        CxGetAsyncErrorTime( pErrStruct ),
        CxGetAsyncErrorFile( pErrStruct, 0 ),
        CxGetAsyncErrorLine( pErrStruct, 0 )
        );
    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++)
    {
        strMsg1.Format("\n\tcalled from line %ld of %s",  
                CxGetAsyncErrorLine( pErrorStruct, i),
                CxGetAsyncErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
```

strMsg1.Format( "\n%d errors have occurred since server connection", CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, MB_ICONEXCLAMATION | MB_OK | MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), CX_MECH_ABORT );
    }

    strMsg.Format("\t\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ",
        CxGetErrorNumber( pErrStruct),
        CxGetErrorConstant (pErrStruct),
        CxGetErrorMsg( pErrStruct ),
        CxGetErrorTime( pErrStruct ),
        CxGetErrorFile( pErrStruct, 0 ),
        CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\n\t\tcalled from line %ld of %s",
            CxGetErrorLine( pErrorStruct, i),
            CxGetErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
    strMsg1.Format( "\n%d errors have occurred since server connection", CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

SEE ALSO
    CxGetErrorAction
**CxGetAsyncErrorConstant**

Gets the error number of the last asynchronous error that occurred in a Cimetryx API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorConstant (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- **PErrorStruct** Pointer to the server asynchronous error structure, received by calling `CxGetError`.

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetryx API call. This number is a constant defined by Cimetryx.

**RETURN VALUES**

Function returns the last error number if successful.

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("%s Error %ld - %s: %s
At:%s
File: %s Line: 
 ",
                CxGetAsyncErrorNumber(pErrStruct),
                CxGetAsyncErrorConstant(pErrStruct),
                CxGetAsyncErrorMsg(pErrStruct),
                CxGetAsyncErrorTime(pErrStruct),
                CxGetAsyncErrorFile(pErrStruct, 0),
                CxGetAsyncErrorLine(pErrStruct, 0));
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTrace; i++ )
        {
            strMsg1.Format("called from line %ld of %s",
                            CxGetAsyncErrorLine(pErrorStruct, i),
                            CxGetAsyncErrorFile(pErrorStruct, i));
            StrMsg += strMsg1;
        }
    }
```
strMsg1.Format( "\n%d errors have occurred since server
connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

    return 0;
} else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
            CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
        CxGetErrorNumber( pErrStruct),
        CxGetErrorConstant (pErrStruct),
        CxGetErrorMsg( pErrStruct ),
        CxGetErrorTime( pErrStruct ),
        CxGetErrorFile( pErrStruct, 0 ),
        CxGetErrorLine( pErrStruct, 0 )
        );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++)
{
    strMsg1.Format("\n\tcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
}
strMsg1.Format( "\n%d errors have occurred since server
connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetErrorConstant
**CxGetAsyncErrorFile**

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorFile (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- `pErrorStruct` Pointer to the server asynchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: \n ",
                        CxGetAsyncErrorNumber( pErrStruct),
                        CxGetAsyncErrorConstant (pErrStruct),
                        CxGetAsyncErrorMsg( pErrStruct ),
                        CxGetAsyncErrorTime( pErrStruct ),
                        CxGetAsyncErrorFile( pErrStruct, 0 ),
                        CxGetAsyncErrorLine( pErrStruct, 0 )
                        );

        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++ )
        {
            strMsg1.Format("\n\tcalled from line %ld of %s",
                        CxGetAsyncErrorLine( pErrorStruct, i),
                        CxGetAsyncErrorFile( pErrorStruct, i));
            StrMsg += strMsg1;
        }
    }
}
```
strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
           MB_ICONEXCLAMATION | MB_OK |
           MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
                               CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ",
              CxGetErrorNumber( pErrStruct),
              CxGetErrorConstant (pErrStruct),
              CxGetErrorMsg( pErrStruct ),
              CxGetErrorTime( pErrStruct ),
              CxGetErrorFile( pErrStruct, 0 ),
              CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\ntcalled from line %ld of %s",
                   CxGetErrorLine( pErrorStruct, i),
                   CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}

strMsg1.Format( "\n%d errors have occurred since server
connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetErrorAction
CxGetAsyncErrorId

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

SYNOPSIS

#include <code\robpac_error.h>
long CxGetAsyncErrorId (CxGetErrorStruct *pErrorStruct)

ARGUMENTS

pErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

DESCRIPTION

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff){
  CxErrorStruct* pErrStruct;
  CString strMsg, strMsg1;
  if( async )
  {
    pErrStruct = CxGetAsyncError( server );
    if( CxGetErrorAction( pErrStruct ) )
    {
      CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
    }
    strMsg.Format("\tError %ld - %s: %s
At:%s
File: %s Line: ",
      CxGetAsyncErrorNumber( pErrStruct),
      CxGetAsyncErrorConstant (pErrStruct),
      CxGetAsyncErrorMsg( pErrStruct ),
      CxGetAsyncErrorTime( pErrStruct ),
      CxGetAsyncErrorFile( pErrStruct, 0 ),
      CxGetAsyncErrorLine( pErrStruct, 0 )
    );
    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
      strMsg1.Format("\ncalled from line %ld of %s",
               CxGetAsyncErrorLine( pErrStruct, i),
               CxGetAsyncErrorFile( pErrStruct, i));
      StrMsg += strMsg1;
    }
  }
}
strMsg1.Format( "\n%d errors have occurred since server connection",
            CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
            MB_ICONEXCLAMATION | MB_OK |
            MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
                                    CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
            CxGetErrorNumber( pErrStruct),
            CxGetErrorConstant (pErrStruct),
            CxGetErrorMsg( pErrStruct ),
            CxGetErrorTime( pErrStruct ),
            CxGetErrorFile( pErrStruct, 0 ),
            CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\t\ntcalled from line %ld of %s",
            CxGetErrorLine( pErrorStruct, i),
            CxGetErrorFile( pErrorStruct, i));
        strMsg += strMsg1;
    }

    strMsg1.Format( "\n%d errors have occurred since server
             connection",
            CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}
}

SEE ALSO

CxGetErrorId
**CxGetAsyncErrorJoint**

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call.

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorJoint (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- **pErrorStruct**: Pointer to the server asynchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful.

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("Error %ld - %s: %s\nAt:%s\nFile: %s Line: ",
                        CxGetAsyncErrorNumber( pErrStruct),
                        CxGetAsyncErrorConstant (pErrStruct),
                        CxGetAsyncErrorMsg( pErrStruct ),
                        CxGetAsyncErrorTime( pErrStruct ),
                        CxGetAsyncErrorFile( pErrStruct, 0 ),
                        CxGetAsyncErrorLine( pErrStruct, 0 )
                        )
        
        lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++ )
        {
            strMsg1.Format("\ntcalled from line %ld of %s",
                        CxGetAsyncErrorLine( pErrorStruct, i),
                        CxGetAsyncErrorFile( pErrorStruct, i));
            StrMsg += strMsg1;
        }
    }
}
```
strMsg1.Format("\n%d errors have occurred since server connection", 
        CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
        MB_ICONEXCLAMATION | MB_OK | 
        MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrStruct ) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrStruct), 
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ",
           CxGetErrorNumber( pErrStruct),
           CxGetErrorConstant (pErrStruct),
           CxGetErrorMsg( pErrStruct ),
           CxGetErrorTime( pErrStruct ),
           CxGetErrorFile( pErrStruct, 0 ),
           CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\ntcalled from line %ld of %s", 
           CxGetErrorLine( pErrorStruct, i),
           CxGetErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server 
        connection", 
        CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg ); 
return 0;
}

SEE ALSO

CxGetErrorJoint
**CxGetAsyncErrorLine**

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call.

### SYNOPSIS

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorLine (CxGetErrorStruct *pErrorStruct)
```

### ARGUMENTS

- **pErrorStruct**  
  Pointer to the server asynchronous error structure, received by calling `CxGetError`.

### DESCRIPTION

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

### RETURN VALUES

Function returns the last error number if successful.

### EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
global long CimetrixExceptionHandler( CxServer server, CxMechanism mech,  
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s %s
At:%s
File: %s Line: 
",  
            CxGetAsyncErrorNumber( pErrStruct),  
            CxGetAsyncErrorConstant (pErrStruct),  
            CxGetAsyncErrorMsg( pErrStruct ),  
            CxGetAsyncErrorTime( pErrStruct ),  
            CxGetAsyncErrorFile( pErrStruct, 0 ),  
            CxGetAsyncErrorLine( pErrStruct, 0 )
        );
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++ )
        {
            strMsg1.Format("\nt-called from line %ld of %s",  
                CxGetAsyncErrorLine( pErrorStruct, i),  
                CxGetAsyncErrorFile( pErrorStruct, i));
            strMsg += strMsg1;
        }  
}```
strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;

else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
    
    strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++ )
        
    strMsg1.Format("\ntcalled from line %ld of %s",
    CxGetErrorLine( pErrorStruct, i),
    CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
    
    strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );
    
    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

SEE ALSO

CxGetErrorLine
CxGetAsyncErrorMech

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

SYNOPSIS

```
#include <code\robpac_error.h>
long CxGetAsyncErrorMech (CxGetErrorStruct *pErrorStruct)
```

ARGUMENTS

*pErrorStruct Pointer to the server asynchronous error struct, received by calling CxGetError

DESCRIPTION

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }

    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
",
        CxGetAsyncErrorNumber( pErrStruct ),
        CxGetAsyncErrorConstant (pErrStruct),
        CxGetAsyncErrorMsg( pErrStruct ),
        CxGetAsyncErrorTime( pErrStruct ),
        CxGetAsyncErrorFile( pErrStruct, 0 ),
        CxGetAsyncErrorLine( pErrStruct, 0 )
    );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\n\t\ncalled from line %ld of %s",
            CxGetAsyncErrorLine( pErrStruct, i),
            CxGetAsyncErrorFile( pErrStruct, i));
        StrMsg += strMsg1;
    }
```
strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\n\t\ntcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format( "\n%d errors have occurred since server
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetErrorMech
CxCGetAsyncErrorMotor

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

SYNOPSIS

#include <code\robpac_error.h>

long CxCGetAsyncErrorMotor (CxGetErrorStruct *pErrorStruct)

ARGUMENTS

pErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

DESCRIPTION

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
", 
            CxGetAsyncErrorNumber( pErrStruct), 
            CxGetAsyncErrorConstant (pErrStruct), 
            CxGetAsyncErrorMsg( pErrStruct ), 
            CxGetAsyncErrorTime( pErrStruct ), 
            CxGetAsyncErrorFile( pErrStruct, 0 ),
            CxGetAsyncErrorLine( pErrStruct, 0 )
        )
    }
    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
        strMsg1.Format("\ntcalled from line %ld of %s", 
            CxGetAsyncErrorLine( errorStruct, i),
            CxGetAsyncErrorFile( errorStruct, i));
        StrMsg += strMsg1;
    }
}
strMsg1.Format( "\n%d errors have occurred since server connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}

else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), 
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\tcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
} 

strMsg1.Format( "\n%d errors have occurred since server 
connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetErrorMotor
**CxGetAsyncErrorMsg**

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorMsg (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

pErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, 
   int tag, int async, char* err_buff) 
{
   CxErrorStruct* pErrStruct;
   CString strMsg, strMsg1;

   if( async )
   {
      pErrStruct = CxGetAsyncError( server );
      if( CxGetErrorAction( pErrStruct ) )
      {
         CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
      }
   }

   strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
",
      CxGetAsyncErrorNumber( pErrStruct),
      CxGetAsyncErrorConstant (pErrStruct),
      CxGetAsyncErrorMsg( pErrStruct ),
      CxGetAsyncErrorTime( pErrStruct ),
      CxGetAsyncErrorFile( pErrStruct, 0 ),
      CxGetAsyncErrorLine( pErrStruct, 0 )
    );

   long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
   for (int i = 1; i < lTtrace; i++)
   {
      strMsg1.Format("\n\t\ncalled from line %ld of %s",
          CxGetAsyncErrorLine( pErrStruct, i),
          CxGetAsyncErrorFile( pErrStruct, i));
      StrMsg += strMsg1;
   }
```
\texttt{strMsg1.Format( "\n%d errors have occurred since server connection", 
         CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
               MB_ICONEXCLAMATION | MB_OK | 
               MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), 
                                    CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s\nAt: %s
File: %s Line: ", 
               CxGetErrorNumber( pErrStruct),
               CxGetErrorConstant (pErrStruct),
               CxGetErrorMsg( pErrStruct ),
               CxGetErrorTime( pErrStruct ),
               CxGetErrorFile( pErrStruct, 0 ),
               CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
        strMsg1.Format("\ntcalled from line %ld of %s", 
                       CxGetErrorLine( pErrorStruct, i),
                       CxGetErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
    strMsg1.Format( "\n%d errors have occurred since server connection", 
         CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

\textbf{SEE ALSO}

  CxGetErrorMsg
CxGetAsyncErrorNumber

Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

SYNOPSIS
#include <code\robpac_error.h>
long CxGetAsyncErrorNumber (CxGetErrorStruct *pErrorStruct)

ARGUMENTS
pErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

DESCRIPTION
This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES
Function returns the last error number if successful

EXAMPLE
This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        strMsg.Format("\tError %ld - %s: %s
At:%s
File: %s Line: 
 ",
            CxGetAsyncErrorNumber( pErrStruct),
            CxGetAsyncErrorConstant (pErrStruct),
            CxGetAsyncErrorMsg( pErrStruct ),
            CxGetAsyncErrorTime( pErrStruct ),
            CxGetAsyncErrorFile( pErrStruct, 0 ),
            CxGetAsyncErrorLine( pErrStruct, 0 )
            );
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++)
            strMsg1.Format("\tcalled from line %ld of %s",
                CxGetAsyncErrorMsg( pErrStruct, i),
                CxGetAsyncErrorFile( pErrStruct, i));
        StrMsg += strMsg1;
    }
strMsg1.Format( "\nThe error %ld - %s: %s
At: %s
File: %s Line: ",
CxGetErrorNumber( pErrStruct),
CxGetErrorConstant (pErrStruct),
CxGetErrorMsg( pErrStruct ),
CxGetErrorTime( pErrStruct ),
CxGetErrorFile( pErrStruct, 0 ),
CxGetErrorLine( pErrStruct, 0 )
);

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTrace; i++ )
{
    strMsg1.Format("\n\ntcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    strMsg1 += strMsg1;
}
strMsg1.Format( "\nThe %d errors have occurred since server
connection",
        CxGetAsyncErrorId( pErrStruct ) );
strMsg += strMsg1;
throw( strMsg );
return 0;
}

SEE ALSO

CxGetErrorNumber
**CxGetAsyncErrorTime**

*Gets the error number of the last asynchronous error that occurred in a Cimetrix API call*

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetAsyncErrorTime (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- **pErrorStruct**  
  Pointer to the server asynchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

- Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
                               int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s At:%s File: %s Line: 
", CxGetAsyncErrorNumber( pErrStruct ), CxGetAsyncErrorConstant (pErrStruct), CxGetAsyncErrorMsg( pErrStruct ), CxGetAsyncErrorTime( pErrStruct ), CxGetAsyncErrorFile( pErrStruct, 0 ), CxGetAsyncErrorLine( pErrStruct, 0 ));

        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTrace; i++ )
        {
            strMsg1.Format("\tcalled from line %ld of %s", CxGetAsyncErrorLine( pErrorStruct, i), CxGetAsyncErrorFile( pErrorStruct, i));
            StrMsg += strMsg1;
        }
    }
```
strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

    return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
            CX_MECH_ABORT );
    }

strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\nt called from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}    strMsg1.Format( "\n%d errors have occurred since server
connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

    CxGetErrorTime
CxGetAsyncErrorTraceCount
Gets the error number of the last asynchronous error that occurred in a Cimetrix API call

SYNOPSIS
#include <code\robpac_error.h>
long CxGetAsyncErrorTraceCount (CxGetErrorStruct *pErrorStruct)

ARGUMENTS
pErrorStruct Pointer to the server asynchronous error structure, received by calling CxGetError

DESCRIPTION
This function returns the error number of the last asynchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES
Function returns the last error number if successful

EXAMPLE
This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
CxErrorStruct* pErrStruct;
CString strMsg, strMsg1;
if( async )
{
pErrStruct = CxGetAsyncError( server );
if( CxGetErrorAction( pErrStruct ) )
{
CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
}
strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: ",
CxGetAsyncErrorNumber( pErrStruct),
CxGetAsyncErrorConstant (pErrStruct),
CxGetAsyncErrorMsg( pErrStruct ),
CxGetAsyncErrorTime( pErrStruct ),
CxGetAsyncErrorFile( pErrStruct, 0 ),
CxGetAsyncErrorLine( pErrStruct, 0 )
);
long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
strMsg1.Format("\ncalled from line %ld of %s",
CxGetAsyncErrorLine( pErrorStruct, i),
CxGetAsyncErrorFile( pErrorStruct, i));
strMsg += strMsg1;
}
strMsg1.Format( "%d errors have occurred since server connection", CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, MB_ICONEXCLAMATION | MB_OK | MB_SETFOREGROUND | MB_TOPMOST );

return 0;
} else {
    pErrStruct = CxGetError( server );
    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ", CxGetErrorNumber( pErrStruct), CxGetErrorConstant (pErrStruct), CxGetErrorMsg( pErrStruct ), CxGetErrorTime( pErrStruct ), CxGetErrorFile( pErrStruct, 0 ), CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\tcalled from line %ld of %s", CxGetErrorLine( pErrorStruct, i), CxGetErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }

    strMsg1.Format( "%d errors have occurred since server connection", CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

SEE ALSO

CxGetErrorTraceCount
**CxGetError**

Gets the structure containing information about the last synchronous error that occurred for the server connection.

**SYNOPSIS**

```c
#include <code/robpac_error.h>
CxErrorStruct *CxGetError (CxServer server)
```

**ARGUMENTS**

- `server` Connection to the current server

**DESCRIPTION**

When a synchronous error occurs in a Cimetrix API call, all of the information about that error is stored in a CxErrorStruct structure that is a member of the server. The members of the structure are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorID</td>
<td>Number of errors that have occurred since the server connection was established</td>
</tr>
<tr>
<td>ErrorNumber</td>
<td>The Cimetrix error number associated with the error</td>
</tr>
<tr>
<td>ErrorConstant</td>
<td>The defined name of the error number</td>
</tr>
<tr>
<td>ErrorTraceCount</td>
<td>Number of items in line and file arrays</td>
</tr>
<tr>
<td>ErrorLine</td>
<td>Line number where error occurs (possible to trace more than just immediate error line)</td>
</tr>
<tr>
<td>ErrorFile</td>
<td>File where error occurred (possible to trace more than just immediate error file)</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>Text explaining the cause of the error</td>
</tr>
<tr>
<td>ErrorMechanism</td>
<td>If the error was related to a mechanism, this field contains a pointer to that mechanism</td>
</tr>
<tr>
<td>ErrorJoint</td>
<td>If the error was related to a specific joint, this field contains the joint number</td>
</tr>
<tr>
<td>ErrorMotor</td>
<td>If the error was related to a specific motor, this field contains the motor number</td>
</tr>
<tr>
<td>ErrorAction</td>
<td>Boolean flag stating whether CxSendMechanismErrorAction needs to be sent or not</td>
</tr>
<tr>
<td>ErrorTime</td>
<td>Time stamp of when the error occurred</td>
</tr>
</tbody>
</table>

Each of these fields can be accessed individually from other API calls. A pointer to this structure is returned from this call.

**RETURN VALUES**

Function returns the CxErrorStruct member of the server for synchronous errors. If no errors have occurred, the structure will be empty.

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct = CxGetError( server );
    if( async )
    {
```
pErrStruct = CxGetAsyncError( server );
if( CxGetErrorAction( pErrStruct ) )
{
    CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
}

CString strMsg;
strMsg.Format("CODE Async error: %s", err_buff);
MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

    return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
            CX_MECH_ABORT );
    }

    CString strTmp;

    strTmp.Format("\tError %ld - %s: %s File: %s Line: %s At: %s",
        CxGetErrorNumber( pErrStruct),
        CxGetErrorConstant (pErrStruct),
        CxGetErrorMsg( pErrStruct ),
        CxGetErrorFile( pErrStruct ),
        CxGetErrorLine( pErrStruct ),
        CxGetErrorTime( pErrStruct ) );

    throw( strTmp );
    return 0;
}

SEE ALSO

    CxGetAsyncError
**CxGetErrorAction**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetErrorAction (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- `pErrorStruct` Pointer to the server synchronous error structure, received by calling `CxGetError`

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, 
    int tag, int async, char* err_buff) 
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }

    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
",
        CxGetAsyncErrorNumber( pErrStruct),
        CxGetAsyncErrorConstant (pErrStruct),
        CxGetAsyncErrorMsg( pErrStruct ),
        CxGetAsyncErrorTime( pErrStruct ),
        CxGetAsyncErrorFile( pErrStruct, 0 ),
        CxGetAsyncErrorLine( pErrStruct, 0 )
        );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
        strMsg1.Format("\n\tcalled from line %ld of %s",
            CxGetAsyncErrorLine( pErrStruct, i),
            CxGetAsyncErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
```
strMsg1.Format("\n%d errors have occurred since server connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK | 
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ", 
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\n\t\ncalled from line %ld of %s", 
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
}
strMsg1.Format( "\n%d errors have occurred since server 
    connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorAction
**CxGetErrorConstant**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call.

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetErrorConstant (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- `pErrorStruct` Pointer to the server synchronous error structure, received by calling CxGetError.

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful.

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CitexExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }
    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: ",
                  CxGetAsyncErrorNumber( pErrStruct),
                  CxGetAsyncErrorConstant (pErrStruct),
                  CxGetAsyncErrorMsg( pErrStruct ),
                  CxGetAsyncErrorTime( pErrStruct ),
                  CxGetAsyncErrorFile( pErrStruct, 0 ),
                  CxGetAsyncErrorLine( pErrStruct, 0 )
                  );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
        strMsg1.Format("\tcalled from line %ld of %s",
                        CxGetAsyncErrorLine( pErrorStruct, i),
                        CxGetAsyncErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
}
strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrStruct),
            CX_MECH_ABORT );
    }

strMsg.Format("\tError %ld - %s: %s\nAt: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 )
);

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTrace; i++ )
{
    strMsg1.Format("\ntcalled from line %ld of %s",
        CxGetErrorLine( pErrStruct, i),
        CxGetErrorFile( pErrStruct, i));
    strMsg += strMsg1;
}

strMsg1.Format( "\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorConstant
**CxGet_Error_File**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGet_Error_File (CxGet_Error_Struct *pErrorStruct)
```

**ARGUMENTS**

- `pErrorStruct` Pointer to the server synchronous error structure, received by calling `CxGet_Error`

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxError_Struct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }

    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
", CxGetAsyncErrorNumber( pErrStruct),
            CxGetAsyncErrorConstant (pErrStruct),
            CxGetAsyncErrorMsg( pErrStruct ),
            CxGetAsyncErrorTime( pErrStruct ),
            CxGetAsyncErrorFile( pErrStruct, 0 ),
            CxGetAsyncErrorLine( pErrStruct, 0 )
    );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++ )
    {
        strMsg1.Format("\ncalled from line %ld of %s",
                        CxGetAsyncErrorMsg( pErrStruct, i),
                        CxGetAsyncErrorMsgFile( pErrStruct, i));
        strMsg += strMsg1;
    }
```
strMsg1.Format("\n%d errors have occurred since server connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
    MB_ICONEXCLAMATION | MB_OK | 
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), 
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ", 
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) 
    );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTrace; i++)
{
    strMsg1.Format("\tcalled from line %ld of %s", 
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server 
    connection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorFile
**CxGetErrorId**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```
#include <code\robpac_error.h>
long CxGetErrorId (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- **pErrorStruct**: Pointer to the server synchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, 
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }
    strMsg.Format("%d - %s: %s
At:%s
File: %s Line: ",
    CxGetAsyncErrorNumber( pErrStruct),
    CxGetAsyncErrorConstant (pErrStruct),
    CxGetAsyncErrorMsg( pErrStruct ),
    CxGetAsyncErrorTime( pErrStruct ),
    CxGetAsyncErrorFile( pErrStruct, 0 ),
    CxGetAsyncErrorLine( pErrStruct, 0 )
    );
    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("called from line %ld of %s",
            CxGetAsyncErrorMsg( pErrStruct, i),
            CxGetAsyncErrorFile( pErrStruct, i));
        StrMsg += strMsg1;
    }
}
```
strMsg1.Format("\n%d errors have occurred since server connection",
            CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
            MB_ICONEXCLAMATION | MB_OK |
            MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
                                CX_MECH_ABORT );
}

strMsg.Format("Error %ld - %s: %s
At: %s
File: %s Line: ",
               CxGetErrorNumber( pErrStruct),
               CxGetErrorConstant (pErrStruct),
               CxGetErrorMsg( pErrStruct ),
               CxGetErrorTime( pErrStruct ),
               CxGetErrorFile( pErrStruct, 0 ),
               CxGetErrorLine( pErrStruct, 0 )
               );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("called from line %ld of %s",
              CxGetErrorLine( pErrorStruct, i),
              CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server connection",
            CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorId
### CxGetLastErrorJoint

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

#### SYNOPSIS

```c
#include <code\robpac_error.h>
long CxGetLastErrorJoint (CxGetErrorStruct *pErrorStruct)
```

#### ARGUMENTS

- **pErrorStruct**  
  Pointer to the server synchronous error structure, received by calling CxGetError

#### DESCRIPTION

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

#### RETURN VALUES

Function returns the last error number if successful

#### EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,  
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }

    strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
",
        CxGetAsyncErrorNumber( pErrStruct),
        CxGetAsyncErrorConstant (pErrStruct),
        CxGetAsyncErrorMsg( pErrStruct ),
        CxGetAsyncErrorTime( pErrStruct ),
        CxGetAsyncErrorFile( pErrStruct, 0 ),
        CxGetAsyncErrorLine( pErrStruct, 0 )
    );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\tcalled from line %ld of %s",
            CxGetAsyncErrorLine( pErrStruct, i),
            CxGetAsyncErrorFile( pErrStruct, i));
        strMsg += strMsg1;
    }
    return strMsg;
}
```
strMsg1.Format("\n%d errors have occurred since server connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\tcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server
connection",
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorJoint
**CxGetErrorLine**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetErrorLine (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

`pErrorStruct` Pointer to the server synchronous error structure, received by calling `CxGetError`

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
   int tag, int async, char* err_buff)
{
   CxErrorStruct* pErrStruct;
   CString strMsg, strMsg1;

   if( async )
   {
      pErrStruct = CxGetAsyncError( server );
      if( CxGetErrorAction( pErrStruct ) )
      {
         CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
      }
   }

   strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
", CxGetAsyncErrorNumber( pErrStruct),
      CxGetAsyncErrorConstant (pErrStruct),
      CxGetAsyncErrorMsg( pErrStruct ),
      CxGetAsyncErrorTime( pErrStruct ),
      CxGetAsyncErrorFile( pErrStruct, 0 ),
      CxGetAsyncErrorLine( pErrStruct, 0 )
    );

   long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
   for (int i = 1; i < lTtrace; i++)
   {
      strMsg1.Format("\tcalled from line %ld of %s",
        CxGetAsyncErrorLine( pErrorStruct, i),
        CxGetAsyncErrorFile( pErrorStruct, i));
      StrMsg += strMsg1;
   }
```
strMsg1.Format("\n%d errors have occurred since server
collection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
}

strMsg.Format("\tError %ld - %s: %s\nAt: %s
File: %s Line: ",
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 )
);

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{
    strMsg1.Format("\ntcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    StrMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server
collection", 
    CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorLine
CxGetErrorLogPolicy

Gets the policy for reporting/logging error conditions

SYNOPSIS

CxGetErrorLogPolicy (CxServer Server, long *mask, char *f_name, long *append)

ARGUMENTS

Server
Server ID. This is a CxServer ID returned from a previous call to CxOpenServer.

mask
This is a bit mask which may contain CX_LOG_TO_TTY or CX_LOG_TO_FILE.

file_name
Returns name of the log file if the CX_LOG_TO_FILE bit is set in the mask.

append
Returns the flag setting which indicates whether policy is set to append to or overwrite the given file (1 for CX_TRUE OR 0 for CX_FALSE) if CX_LOG_TO_FILE bit is set in the mask.

DESCRIPTION

This command gets the error logging policy. Valid return values are defined as follows:

CX_LOG_TO_TTY
The error message will be reported to the tty window from which the CODE application process is initiated. The default mask is CX_LOG_TO_TTY.

CX_LOG_TO_FILE
The error message will be logged to the file that is specified by file_name argument.

These options may both be set simultaneously.
New error messages will be appended to an existing file if the append flag is set to any non-zero value, and the named file already exists. If the append flag is set to zero then the given file will be overwritten.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_FILE_OPEN_ERROR</td>
<td>The given file cannot be opened.</td>
</tr>
</tbody>
</table>

EXAMPLE

The following program opens a server and a mechanism. It sets the error policy and error log policy to write errors to a file:

```c
void main (void)
{
```
CxServer Server;
CxMechanism mech;
char *error_file[CX_MAX_FILE_PATH];
long error_policy, append, mask;
CxGetErrorLogPolicy (Server, mask, errorfile, append);

SEE ALSO

CxSetErrorLogPolicy
CxGetErrorMech

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

SYNOPSIS

#include <code\robpac_error.h>
long CxGetErrorMech (CxGetErrorStruct *pErrorStruct)

ARGUMENTS

pErrorStruct  Pointer to the server synchronous error structure, received by calling CxGetError

DESCRIPTION

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
   // int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }
    strMsg.Format("\tError %ld - %s: %s\nAt:%s
File: %s Line: 
 ",
                 CxGetAsyncErrorNumber( pErrStruct),
                 CxGetAsyncErrorConstant (pErrStruct),
                 CxGetAsyncErrorMsg( pErrStruct ),
                 CxGetAsyncErrorTime( pErrStruct ),
                 CxGetAsyncErrorFile( pErrStruct, 0 ),
                 CxGetAsyncErrorLine( pErrStruct, 0 )
                 );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++ )
    {
        strMsg1.Format("\ntcalled from line %ld of %s",
                        CxGetAsyncErrorLine( pErrStruct, i),
                        CxGetAsyncErrorFile( pErrStruct, i));
        StrMsg += strMsg1;
    }
}
```csharp
strMsg1.Format( "\n%d errors have occurred since server
collection", 
CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
MB_ICONEXCLAMATION | MB_OK |
MB_SETFOREGROUND | MB_TOPMOST );

return 0;
} 

else 
{

pErrStruct = CxGetError( server );

if( CxGetErrorAction( pErrorStruct) )
{

CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
CX_MECH_ABORT );
}

strMsg.Format( "\tError %ld - %s: %s
At: %s
File: %s Line: ",
CxGetErrorNumber( pErrStruct),
CxGetErrorConstant (pErrStruct),
CxGetErrorMsg( pErrStruct ),
CxGetErrorTime( pErrStruct ),
CxGetErrorFile( pErrStruct, 0 ),
CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
{

strMsg1.Format("\ntcalled from line %ld of %s", 
CxGetErrorLine( pErrorStruct, i),
CxGetErrorFile( pErrorStruct, i));
StrMsg += strMsg1;
}

strMsg1.Format( "\n%d errors have occurred since server
connection", 
CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorMech
```
CxGetErrorMotor

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

SYNOPSIS

```c
#include <code\robpac_error.h>
long CxGetErrorMotor (CxGetErrorStruct *pErrorStruct)
```

ARGUMENTS

- `pErrorStruct` Pointer to the server synchronous error structure, received by calling `CxGetError`

DESCRIPTION

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech,
    int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s
At:%s
File: %s Line: 
 ",
            CxGetAsyncErrorNumber( pErrStruct),
            CxGetAsyncErrorConstant (pErrStruct),
            CxGetAsyncErrorMsg( pErrStruct ),
            CxGetAsyncErrorTime( pErrStruct ),
            CxGetAsyncErrorFile( pErrStruct, 0 ),
            CxGetAsyncErrorLine( pErrStruct, 0 )
        );
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++ )
        {
            strMsg1.Format("\n\t\called from line %ld of %s",
                CxGetAsyncErrorLine( pErrStruct, i),
                CxGetAsyncErrorFile( pErrStruct, i));
            strMsg += strMsg1;
        }
    }
    ```
strMsg1.Format("\n%d errors have occurred since server connection", 
        CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
            MB_ICONEXCLAMATION | MB_OK | 
            MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrStruct ) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrStruct), 
                                      CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ", 
                  CxGetErrorNumber( pErrStruct), 
                  CxGetErrorConstant( pErrStruct), 
                  CxGetErrorMsg( pErrStruct ), 
                  CxGetErrorTime( pErrStruct ), 
                  CxGetErrorFile( pErrStruct, 0 ), 
                  CxGetErrorLine( pErrStruct, 0 ) );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\tcalled from line %ld of %s", 
                        CxGetErrorLine( pErrorStruct, i), 
                        CxGetErrorFile( pErrorStruct, i));
        strMsg += strMsg1;
    }
    strMsg1.Format("\n%d errors have occurred since server connection", 
              CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

SEE ALSO

CxGetAsyncErrorMotor
CxGetErrorMsg

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

SYNOPSIS

#include <code/robpac_error.h>
long CxGetErrorMsg (CxGetErrorStruct *pErrorStruct)

ARGUMENTS

pErrorStruct Pointer to the server synchronous error structure, received by calling CxGetError

DESCRIPTION

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

RETURN VALUES

Function returns the last error number if successful

EXAMPLE

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if (async)
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line:
", CxGetAsyncErrorNumber( pErrStruct), CxGetAsyncErrorConstant (pErrStruct), CxGetAsyncErrorMsg( pErrStruct ), CxGetAsyncErrorTime( pErrStruct ), CxGetAsyncErrorFile( pErrStruct, 0 ), CxGetAsyncErrorLine( pErrStruct, 0 ));
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTrace; i++)
        {
            strMsg1.Format("\tcalled from line %ld of %s", CxGetAsyncErrorLine( pErrStruct, i), CxGetAsyncErrorFile( pErrStruct, i));
            strMsg += strMsg1;
        }
    }
}
strMsg1.Format( "\n%d errors have occurred since server connection",
CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
  pErrStruct = CxGetError( server );

  if( CxGetErrorAction( pErrorStruct ) )
  {
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
        CX_MECH_ABORT );
  }

  strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
      CxGetErrorNumber( pErrStruct),
      CxGetErrorConstant (pErrStruct),
      CxGetErrorMsg( pErrStruct ),
      CxGetErrorTime( pErrStruct ),
      CxGetErrorFile( pErrStruct, 0 ),
      CxGetErrorLine( pErrStruct, 0 ) );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++)
{
  strMsg1.Format("\ntcalled from line %ld of %s",
      CxGetErrorLine( pErrorStruct, i),
      CxGetErrorFile( pErrorStruct, i));
  StrMsg += strMsg1;
}

strMsg1.Format( "\n%d errors have occurred since server
connection",
CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorMsg
**CxGetErronumber**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call.

**SYNOPSIS**
```
#include <code\robpac_error.h>
long CxGetErrorNumber (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**
- **pErrorStruct** Pointer to the server synchronous error structure, received by calling CxGetError

**DESCRIPTION**
This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**
Function returns the last error number if successful.

**EXAMPLE**
This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;

    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }

    strMsg.Format("\tError %ld - %s: %s
At:%s
File: %s Line: 
 ",
    CxGetAsyncErrorNumber( pErrStruct),
    CxGetAsyncErrorConstant (pErrStruct),
    CxGetAsyncErrorMsg( pErrStruct ),
    CxGetAsyncErrorTime( pErrStruct ),
    CxGetAsyncErrorFile( pErrStruct, 0 ),
    CxGetAsyncErrorLine( pErrStruct, 0 )
    );

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++)
    {
        strMsg1.Format("\ntcalled from line %ld of %s",
            CxGetAsyncErrorLine( pErrStruct, i),
            CxGetAsyncErrorFile( pErrStruct, i));
        StrMsg += strMsg1;
    }
```

strMsg1.Format( "\n%d errors have occurred since server
connection", 
        CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,
             MB_ICONEXCLAMATION | MB_OK |
             MB_SETFOREGROUND | MB_TOPMOST );

return 0;
} else
{

  pErrStruct = CxGetError( server );

  if( CxGetErrorAction( pErrorStruct) )
  {
    CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct),
          CX_MECH_ABORT );
  }


strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ",
            CxGetErrorNumber( pErrStruct),
            CxGetErrorConstant (pErrStruct),
            CxGetErrorMsg( pErrStruct ),
            CxGetErrorTime( pErrStruct ),
            CxGetErrorFile( pErrStruct, 0 ),
            CxGetErrorLine( pErrStruct, 0 )
        );

long lTrace = CxGetErrorTraceCount( pErrStruct );
for (int i = 1; i < lTtrace; i++ )
  {
    strMsg1.Format("\n\tcalled from line %ld of %s",
        CxGetErrorLine( pErrorStruct, i),
        CxGetErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
  }

strMsg1.Format( "\n%d errors have occurred since server
connection", 
        CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

throw( strMsg );
return 0;
}

SEE ALSO

CxGetAsyncErrorNumber
**CxGetErrorPolicy**

Gets current error policy for handling error conditions

**SYNOPSIS**

```c
#include <code/robpac.h>
void CxGetErrorPolicy (CxServer Server, long *policy)
```

**ARGUMENTS**

- **Server**: This is a CxServer ID returned from a previous call to CxOpenServer.
- **policy**: The set error handling policy (see possible arguments in the description below).

**DESCRIPTION**

This function gets the current error handling policy. Valid policies are defined as follows:

- **CX_EXIT_ON_ERROR**: When any error occurs, exit the process. All motion associated with this process will be halted.
- **CX_RETURN_ERRORS**: Return all normal errors. Asynchronous errors will cause the process to exit.
- **CX_USER_HANDLER**: When any error occurs, invoke a user-defined error handling function specified by CxAssignErrorHandler. If a user-assigned error handler is not defined, a default error handler is called.
- **CX_USER_ASYNC_HANDLER**: When an asynchronous error occurs, invoke a user-defined error handling function specified by CxAssignErrorHandler. If a user-assigned error handler is not defined, a default error handler is called (if a normal error occurs, an error condition will be returned).

Normal errors occur when the CODE API function called receives a response from the CIMServer indicating an error condition has occurred, or when one of the parameters is incorrect. Asynchronous errors occur when the CIMServer detects an unexpected error and sends an asynchronous error message.

**RETURN VALUES**

This function does not return any values.

**EXAMPLE**

```c
void main (void)
{
    CxServer Server;
    CxMechanism mech;
    long error_policy;

    /*open server*/
    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    CxSetErrorPolicy (Server, CX_RETURN_ERRORS);
    .
    .
    CxGetErrorPolicy (Server, &error_policy);
}
```

**SEE ALSO**

CxAssignErrorHandler, CxSetErrorJump, CxSetErrorPolicy
**CxGetErrorTime**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetErrorTime (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- `pErrorStruct`: Pointer to the server synchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

- Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
    }
    strMsg.Format("\tell %ld - %s\nAt: %s\nFile: %s Line: "
                   , CxGetAsyncErrorNumber( pErrStruct ),
                   CxGetAsyncErrorConstant( pErrStruct ),
                   CxGetAsyncErrorMsg( pErrStruct ),
                   CxGetAsyncErrorFile( pErrStruct, 0 ),
                   CxGetAsyncErrorLine( pErrStruct, 0 ));

    long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTrace; i++ )
    {
        strMsg1.Format("\ntcalled from line %ld of %s",
                       CxGetAsyncErrorLine( pErrStruct, i),
                       CxGetAsyncErrorFile( pErrStruct, i));
        strMsg += strMsg1;
    }
}
```
strMsg1.Format("\n%d errors have occurred since server
collection", 
CxGetAsyncErrorId( pErrStruct ) );

strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg, 
    MB_ICONEXCLAMATION | MB_OK |
    MB_SETFOREGROUND | MB_TOPMOST );

    return 0;
}
else
{
    pErrStruct = CxGetError( server );

    if( CxGetErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxGetErrorMech( pErrorStruct), 
            CX_MECH_ABORT );
    }

    strMsg.Format("\tError %ld - %s: %s
At: %s
File: %s Line: ", 
    CxGetErrorNumber( pErrStruct),
    CxGetErrorConstant (pErrStruct),
    CxGetErrorMsg( pErrStruct ),
    CxGetErrorTime( pErrStruct ),
    CxGetErrorFile( pErrStruct, 0 ),
    CxGetErrorLine( pErrStruct, 0 )
    );

    long lTrace = CxGetErrorTraceCount( pErrStruct );
    for (int i = 1; i < lTtrace; i++ )
    {
        strMsg1.Format("\t\t\tcalled from line %ld of %s", 
            CxGetErrorLine( pErrorStruct, i),
            CxGetErrorFile( pErrorStruct, i));
        StrMsg += strMsg1;
    }
    strMsg1.Format("\n%d errors have occurred since server
connection", 
    CxGetAsyncErrorId( pErrStruct ) );

    strMsg += strMsg1;

    throw( strMsg );
    return 0;
}

SEE ALSO
    CxGetAsyncErrorTime
**CxGetErrorTraceCount**

Gets the error number of the last synchronous error that occurred in a Cimetrix API call

**SYNOPSIS**

```c
#include <code\robpac_error.h>
long CxGetErrorTraceCount (CxGetErrorStruct *pErrorStruct)
```

**ARGUMENTS**

- pErrorStruct  Pointer to the server synchronous error structure, received by calling CxGetError

**DESCRIPTION**

This function returns the error number of the last synchronous error to occur in a Cimetrix API call. This number is a constant defined by Cimetrix.

**RETURN VALUES**

Function returns the last error number if successful

**EXAMPLE**

This example shows how an exception handler checks for a synchronous error versus an asynchronous error and throws a string containing all of the information for the error.

```c
long CimetrixExceptionHandler( CxServer server, CxMechanism mech, int tag, int async, char* err_buff)
{
    CxErrorStruct* pErrStruct;
    CString strMsg, strMsg1;
    if( async )
    {
        pErrStruct = CxGetAsyncError( server );
        if( CxGetErrorAction( pErrStruct ) )
        {
            CxSendMechanismErrorAction( mech, CX_MECH_ABORT_BLOCK );
        }
        strMsg.Format("\tError %ld - %s: %s\nAt:%s\nFile: %s Line: 
", CxGetAsyncErrorNumber( pErrStruct), CxGetAsyncErrorConstant (pErrStruct), CxGetAsyncErrorMsg( pErrStruct ), CxGetAsyncErrorTime( pErrStruct ), CxGetAsyncErrorFile( pErrStruct, 0 ), CxGetAsyncErrorLine( pErrStruct, 0 ) );
        long lTrace=CxGetAsyncErrorTraceCount( pErrStruct );
        for (int i = 1; i < lTtrace; i++)
        {
            strMsg1.Format("\n\tcalled from line %ld of %s", CxGetAsyncErrorLine( pErrStruct, i), CxGetAsyncErrorFile( pErrStruct, i));
            StrMsg += strMsg1;
        }
```
strMsg1.Format("\n%d errors have occurred since server connection",  
    CxAsyncErrorId( pErrStruct ));
strMsg += strMsg1;

MessageBox( GetDesktopWindow(), strMsg,  
    MB_ICONEXCLAMATION | MB_OK |  
    MB_SETFOREGROUND | MB_TOPMOST );

return 0;
}
else
{
    pErrStruct = CxError( server );

    if( CxErrorAction( pErrorStruct) )
    {
        CxSendMechanismErrorAction( CxErrorMech( pErrorStruct),  
            CX_MECH_ABORT );
    }

strMsg.Format("\tError %ld - %s: %s\nAt: %s\nFile: %s Line: ",  
    CxErrorNumber( pErrStruct),  
    CxErrorConstant (pErrStruct),  
    CxErrorMsg( pErrStruct ),  
    CxErrorTime( pErrStruct ),  
    CxErrorFile( pErrStruct, 0 ),  
    CxErrorLine( pErrStruct, 0 ) );

long lTrace = CxErrorTraceCount( pErrStruct );
for (int i = 1; i < lTrace; i++ )
{
    strMsg1.Format("\n\tcalled from line %ld of %s",  
        CxErrorLine( pErrorStruct, i),  
        CxErrorFile( pErrorStruct, i));
    strMsg += strMsg1;
}
strMsg1.Format("\n%d errors have occurred since server connection",  
    CxAsyncErrorId( pErrStruct ));
strMsg += strMsg1;
throw( strMsg );
return 0;
}

SEE ALSO

CxAsyncErrorTraceCount
**CxSendMechanismErrorAction**

Specifies action to take after motion is abnormally halted

**SYNOPSIS**

```c
#include <code/robpac.h>
long CxSendMechanismErrorAction (CxMechanism mech, long action) ;
```

**ARGUMENTS**

- `mech` Mechanism ID;
  This is a CxMechanism ID returned by a call to CxOpenMechanism.

- `action` Action to be taken; see **DESCRIPTION** below for possible error actions.

**DESCRIPTION**

Any time a mechanism’s motion is terminated abnormally, this function must be called before the controller will allow the mechanism to move. There are several ways in which motion can be abnormally terminated. An error may occur such as a command to move beyond an axis limit, a servo following error, the operator may press the E-Stop Button, a registered I/O event may halt motion, or the application may make a call to CxStopMotion.

This function tells the controller how to resume processing motion commands. The action may be to resume the current move, start the next move from the halted location, or flush all pending moves. Any moves sent to the controller after motion is halted will be added to the queue of pending moves. If this queue fills, the next move sent will block. Since motion processing is halted, the queue will not drain until CxSendMechanismErrorAction is called, which can lead to potential deadlock in single threaded applications. To prevent this, exception handlers should call CxSendMechanismErrorAction before sending additional moves. Multi-threaded applications may continue to send moves to the controller as long as an exception handling thread is available to resume motion. A description of the possible actions follows:

- **CX_MECH_ABORT_BLOCK** Abort the current move in the motion queue, flush the remaining moves in the queue and also block the motion queue. The user will need to call CxSendMechanismErrorAction with CX_MECH_ABORT to unblock the queue.

- **CX_MECH_RESUME** Resume the current move in the motion queue and continue processing subsequent moves in the queue. This action is not allowed after an E-Stop.

- **CX_MECH_ABORT** Abort the current move in the motion queue and flush the remaining moves in the queue.

- **CX_MECH_NEXT** Abort the current move in the motion queue, and process moves starting with the next move in the motion queue. If the current move or the next move in the queue are path moves, if the next move is a CX_CIRCULAR_INTERP move, or if the current move is an CX_END_POINT_MOVE, this function will return an error condition.

If this function is called when no motion error or halt condition exists, it returns immediately with no consequences.
WARNINGS

If CxSendMechanismErrorAction returns an error, calling other move related functions will eventually fill the move queue and will cause the process to hang indefinitely. These functions include all move API functions, CxSetServerType, CxUpdateJoints, CxWaitForEndOfMotion.

If CxSendMechanismErrorAction returns an error because of an amp fault, calling CxStopMechanism again will cause the process to hang if the move was a PVT move. The solution is to re-enable the amps, if necessary. Then, call CxSendMechanismErrorAction again, after the error condition has been fixed.

RETURN VALUES

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

ERRORS

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_MESSAGE_SEND_FAILED</td>
<td>Error in sending message.</td>
</tr>
<tr>
<td>CX_MESSAGE_RECEIVED_FAILED</td>
<td>Error in receiving message.</td>
</tr>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The parameter for the action variable was invalid.</td>
</tr>
<tr>
<td>CX_NO_MECH_NEXT_ON_END_POINT_MOVE</td>
<td>The error action is defined as CX_MECH_NEXT, and the current move in the queue has a trajectory_type of CX_END_POINT_MOVE.</td>
</tr>
<tr>
<td>CX_NO_MECH_NEXT_ON_CIRC_INTERP</td>
<td>The error action is defined as CX_MECH_NEXT, and the next move in the queue has a CIRCULAR interpolation type.</td>
</tr>
<tr>
<td>CX_NO_MECH_NEXT_ON_PATH</td>
<td>The error action is defined as CX_MECH_NEXT, and the current move or the next move in the CIMSServer motion queue is associated with a CIMSServer path segment.</td>
</tr>
<tr>
<td>CX_TELE_PARM_NOT_SET</td>
<td>The error action is defined as CX_MECH_TELE, and the tele_Parms have not been set (see CxSetTeleModeParm).</td>
</tr>
<tr>
<td>CX_MECH_NOT_OPEN_FOR_CONTROL</td>
<td>Mechanism is not open for control.</td>
</tr>
</tbody>
</table>

EXAMPLES

The following example demonstrates aborting a motion after calling CxStopMechanism.

```c

void main (void)
{
    CxServer Server ;
    CxMechanism mech ;
    CxNodeId targ_id, tcf_id, mech_id ;
```
move_to_node (Mech, targ_id, tcf_id);
.

SEE ALSO

CxSetTeleModeParm, CxStopMechanism
**CxSetErrorLogPolicy**

Sets the policy for reporting/logging error conditions

**SYNOPSIS**

```c
#include <code/robpac.h>
void CxErrorLogPolicy (CxServer Server, long mask, char *f_name, long append)
```

**ARGUMENTS**

- **Server**
  This is a CxServer ID returned from a previous call to CxOpenServer
- **mask**
  A bit-mapped mask that determines how the error message is logged. If “0” is entered as the mask, then error logging is turned off.
- **f_name**
  Name of the log file if CX_LOG_TO_FILE bit is set
- **append**
  Flag indicates whether to append to or overwrite the given file (CX_TRUE OR CX_FALSE) if CX_LOG_TO_FILE bit is set

**DESCRIPTION**

This function sets the error logging policy based on the given mask. Valid options for the `mask` are defined as follows:

- **CX_LOG_TO_TTY**
  The error message will be reported to the tty window from which the CODE application process is initiated. The default mask is CX_LOG_TO_TTY.
- **CX_LOG_TO_FILE**
  The error message will be logged to the file that is specified by `f_name` argument.

These options can be OR-ed together to form the mask (see example below).
New error messages will be appended to an existing file if the `append` flag is set to any non-zero value, and the named file already exists. If the `append` flag is set to zero then the given file will be overwritten.

**RETURN VALUES**

This function returns **0** if successful; otherwise, **-1** (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_FILE_OPEN_ERROR</td>
<td>The given file cannot be opened. Usually this is because the application process does not have proper write permissions for the specified file.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following program opens a server and a mechanism. It sets the error policy and error log policy to write errors simultaneously to the screen and to a file:

#include <stdio.h>
#include <code/robpac.h>

void main (void)
{
    CxServer    Server;
    CxMechanism mech;
    CxNodeId   s100;
    long        jnt_nr;

    Server = CxOpenServer ("Testing", CX_SMEM, 0);
    CxSetErrorPolicy (Server, CX_RETURN_ERRORS);
    CxSetErrorLogPolicy (CX_LOG_TO_FILE|CX_LOG_TO_TTY, "error.log",
                         CX_FALSE);
.
.
}

SEE ALSO

    CxSetErrorPolicy
**CxSetErrorPolicy**

Sets the policy for handling exceptions

**SYNOPSIS**

```
#include <code/robpac.h>
long CxSetErrorPolicy (CxServer Server, long policy)
```

**ARGUMENTS**

- **Server**
  This is a CxServer ID returned from a previous call to CxOpenServer.

- **policy**
  The desired error handling policy (see possible arguments in the description below).

**DESCRIPTION**

This function sets the exception handling policy. Valid policies are defined as follows:

- **CX_EXIT_ON_ERROR**
  When any error occurs, exit the process by calling CxRobpacExit.

- **CX_RETURN_ERRORS**
  Return all normal errors. Asynchronous errors will cause the process to exit.

- **CX_USER_HANDLER**
  When any error occurs, invoke a user-defined error handling function specified by CxAssignErrorHandler. If a user-assigned error handler is not defined, a default error handler is called.

- **CX_USER_ASYNC_HANDLER**
  When an asynchronous error occurs, invoke a user-defined error handling function specified by CxAssignErrorHandler. If a user-assigned error handler is not defined, a default error handler is called. If a normal error occurs, return an error condition.

- **CX_THROW_ERROR**
  When an asynchronous error occurs, throw a C++ exception error of the type CxErrorStruct.

Normal errors occur when the CODE API function called receives a response from the CIMServer indicating an error condition has occurred, or when one of the functional parameters is incorrect. Asynchronous errors occur when the CIMServer detects an unexpected exception and sends a signal to the CODE application process, followed by an asynchronous error message.

**RETURN VALUES**

This function returns 0 if successful; otherwise, -1 (CX_ERROR) is returned.

**ERRORS**

If the function returns an error condition, the error code can be obtained by using the CxGetErrorNumber function. The possible error codes are defined in the following table:

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CX_INVALID_ARGUMENT</td>
<td>The policy passed in is not valid.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

The following examples illustrate how to set error policy for different conditions:
Set error policy to return on errors.

```c
void main (void)
{
    CxServer Server ;
    .
    Server = CxOpenServer ("MyServer", CX_SMEM, 0);
    If (Server == 0) {
        CxRobpacExit();
        CxSetErrorPolicy (Server, CX_RETURN_ERRORS) ;
    }.
}
```

Set error policy to exit on errors (this is the default).

```c
void main (void)
{
    CxServer Server ;
    .
    Server = CxOpenServer ("MyServer", CX_SMEM, 0) ;
    .
    CxSetErrorPolicy (Server, CX_EXIT_ON_ERROR) ;
}
```

Set error policy to invoke a user-defined error handler on all errors.

```c
long MyHandler (CxServer server, CxMechanism mech, int tag, int async,
                char *err_buff);

void main (void)
{
    CxServer Server ;
    long my_handler () ;
    .
    CxAssignErrorHandler (Server, MyHandler) ;
    .
    Server = CxOpenServer ("MyServer", CX_SMEM, 0) ;
    CxSetErrorPolicy (Server, CX_USER_HANDLER) ;
    .
}
```

```c
long MyHandler (CxServer server, CxMechanism mech, int tag, int async,
                char *err_buff)
{
    fprintf (stderr,"I got an error with code %ld\n",
             CxGetErrorNumber(server)) ;
    fprintf ( stderr, "%s\n", err_buff) ;
}
```

This example shows how `CX_THROW_ERROR` is used.

```c
Void main ( )
{
    .
}
try {
    CxMoveSingleAxis(mech, 0, ...);
}

SEE ALSO

CxAssignErrorHandler, CxSetErrorJump, CxGetErrorPolicy