Sample Applications

--Table of Contents--

adc1.c -- Read the Analog to Digital converter.

```
/* adc1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/adc1.c 10    8/01/01 2:08p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    adc1Main

argMainRENAME(main, adc1)
#endif

/*

:Read the Analog to Digital converter.

This sample code demonstrates how to read digital values from the A/D (analog
 to digital) converter.  The values are stored into a buffer and the average
 of the values stored in the buffer is displayed.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define ADC_COUNT   (1)
#define ADC_NUMBER  (0)
#define ADC_RANGE   (10.0)  /* 10.0, 5.0, 2.5, or 1.25 volts */
#define ADC_INPUT   (MEIAdcMuxANALOG_IN_0)

#define BUFFER_SIZE (1000)
```

```
/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIAdc             *adc,
                 long                adcNumber)
{
    long           returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create adc object */
    *adc =
        mpiAdcCreate(*control,
                     adcNumber);
    msgCHECK(mpiAdcValidate(*adc));
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIAdc      *adc)
{
```

```
    long    returnValue;


    /* Delete adc object */
    returnValue =
        mpiAdcDelete(*adc);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Enable adcs */
void enableAdcs(MPIControl  control,
                long        adcCount)
{
    MPIControlConfig    controlConfig;
    long                returnValue;


    /* Read controller configuration */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    controlConfig.adcCount = adcCount;

    /* Write controller configuration */
    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);
}


/* Configure adc object */
void adcConfigure(MPIAdc    adc,
                  MEIAdcMux input,
                  double    range)
{
    MPIAdcConfig    adcConfig;
    MEIAdcConfig    adcConfigXmp;
    long            returnValue;


    /* Read adc configuration */
    returnValue =
        mpiAdcConfigGet(adc,
```

```
                        &adcConfig,
                        &adcConfigXmp);
    msgCHECK(returnValue);

    /* Set voltage range */
    adcConfig.range = range;

    /* Set voltage input */
    adcConfigXmp.mux = input;

    /* Write adc configuration */
    returnValue =
        mpiAdcConfigSet(adc,
                        &adcConfig,
                        &adcConfigXmp);
    msgCHECK(returnValue);
}


/* Display ADC input values */
void displayAdcInput(MPIAdc  adc,
                     long   *buffer,
                     long    bufferSize)
{
#define DISPLAY_REFRESH (50)    /* To avoid console flicker */

    MPIAdcConfig    adcConfig;

    double  adcRange;
    double  adcMean;
    double  sumBuffer   = 0.0;
    long    bufferIndex = 0;
    long    averageSize = 0;
    long    returnValue;


    fprintf(stderr, "Press any key to quit...\n\n");

    /* Read adc configuration */
    returnValue =
        mpiAdcConfigGet(adc,
                        &adcConfig,
                        NULL);
    msgCHECK(returnValue);

    /* Set voltage range */
    adcRange = adcConfig.range;

    while (meiPlatformKey(MPIWaitPOLL) < 0) {

        /* Read ADC input value */
        returnValue =
            mpiAdcInput(adc,
                        (unsigned long*)&buffer[bufferIndex]);
```

```
        msgCHECK(returnValue);

        sumBuffer+=buffer[bufferIndex];
        bufferIndex++;
        bufferIndex%=bufferSize;

        if (averageSize < bufferSize) {
            averageSize++;
            adcMean = sumBuffer / averageSize;
        }
        else {
            adcMean = sumBuffer / averageSize;
            sumBuffer-=buffer[bufferIndex];
        }

        if (bufferIndex%DISPLAY_REFRESH == 0) {

            /* Display mean adc input value */
            fprintf(stderr,
                    "Mean of the last %d ADC reads = %lf Volts.   \r",
                    averageSize,
                    adcMean * adcRange / 32767.0);
        }

        /* Wait one controller sample */
        meiControlSampleWait(mpiAdcControl(adc),
                             1);
    }

    fprintf(stderr, "\n\n");

#undef DISPLAY_REFRESH
}


int main(int     argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIAdc              adc;

    long    adcBuffer[BUFFER_SIZE];


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
```

```
                controlType,
                &controlAddress,
                &adc,
                ADC_NUMBER);

    /* Enable adcs */
    enableAdcs(control,
               ADC_COUNT);

    /* Configure adc object */
    adcConfigure(adc,
                 ADC_INPUT,
                 ADC_RANGE);

    /* Display ADC input values */
    displayAdcInput(adc,
                    adcBuffer,
                    BUFFER_SIZE);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &adc);

        return (MPIMessageOK);
}
```

adc2.c -- Determine scale and offset calibration for Analog to Digital converter input.

```
/* adc2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/adc2.c 5     8/01/01 2:08p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    adc2Main

argMainRENAME(main, adc2)
#endif

/*

:Determine scale and offset calibration for Analog to Digital converter input.

This sample code demonstrates how to determine calibration factors for
 an ADC input channel.  This sample requires an external source to generate
 the analog voltage into an ADC channel.

The calibration factors are determined by sampling two known voltages.
 The default voltages are 5 volts and -5 volts.  Other voltages can be set by
 changing the definitions of the #define's LINE_VOLTAGE_1 and LINE_VOLTAGE_2.
 This default input lines are Analog_IN_0 and Analog_IN_1.  These can be changed
 by changing the definitions of the #define's LINE_INPUT_1 and LINE_INPUT_2.

ATTENTION: The uncertanties of the input voltage lines should also be specified
 for an accurate analysis.  The smaller these uncertainties, the better analysis
 you will receive.  The default values are 0.01 Volts and can be specified by
 changing the #define's LINE_VOLTAGE_UNCERTAINTY_1 and
 LINE_VOLTAGE_UNCERTAINTY_2.

Using the ADC values, voltage values, and uncertainties, displayAdcAnalysis()
 determines the linear "scale" factor (m) and the "offset" (b).  These
 calibration factors can be applied to future ADC value conversions in
 application code, to improve the ADC accuracy and analysis of the values read.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define ADC_COUNT       (8)

#define ADC_RANGE                   (10.0)  /* 10.0, 5.0, 2.5, or 1.25 volts */
#define LINE_VOLTAGE_1              (5.00)
#define LINE_VOLTAGE_2              (-5.00)
#define LINE_VOLTAGE_UNCERTAINTY_1  (0.01)
#define LINE_VOLTAGE_UNCERTAINTY_2  (0.01)

#define LINE_INPUT_1        (MEIAdcMuxANALOG_IN_0)
#define LINE_INPUT_2        (MEIAdcMuxANALOG_IN_1)

#define SAMPLES     (1000)  /* Samples per input line */


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIAdc             *adc,
                 long                adcCount)
{
    long index;
    long returnValue;


    /* Create motion controller object */
    *control =
```

```
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create adc objects */
    for (index = 0; index < adcCount; index++) {
        adc[index] =
            mpiAdcCreate(*control,
                         index);
        msgCHECK(mpiAdcValidate(adc[index]));
    }
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIAdc      *adc,
                    long         adcCount)
{
    long index;
    long returnValue;


    /* Delete adc objects */
    for (index = 0; index < adcCount; index++) {
        returnValue =
            mpiAdcDelete(adc[index]);
        msgCHECK(returnValue);
    }

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Enable adcs */
void enableAdcs(MPIControl  control,
                long        adcCount)
{
    MPIControlConfig    controlConfig;
    long                returnValue;


    /* Read controller configuration */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);
```

```
    controlConfig.adcCount = adcCount;

    /* Write controller configuration */
    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);
}


/* Configure adc object */
void adcConfigure(MPIAdc    adc,
                  MEIAdcMux input,
                  double    range)
{
    MPIAdcConfig    adcConfig;
    MEIAdcConfig    adcConfigXmp;
    long            returnValue;


    /* Read adc configuration */
    returnValue =
        mpiAdcConfigGet(adc,
                        &adcConfig,
                        &adcConfigXmp);
    msgCHECK(returnValue);

    /* Set voltage range */
    adcConfig.range = range;

    /* Set voltage input */
    adcConfigXmp.mux = input;

    /* Write adc configuration */
    returnValue =
        mpiAdcConfigSet(adc,
                        &adcConfig,
                        &adcConfigXmp);
    msgCHECK(returnValue);
}


/* Calculate stastical input information */
void getAdcInputInfo(MPIAdc *adc,
                     long    adcCount,
                     long    samples,
                     double *inputMean,
                     double *inputVar)
{
    MPIAdcConfig    adcConfig;

    double sumInput   = 0.0;
    double sumInputSq = 0.0;
```

```
    long input;
    long index;
    long returnValue;


    /* Read adc configuration */
    returnValue =
        mpiAdcConfigGet(adc[0],
                        &adcConfig,
                        NULL);
    msgCHECK(returnValue);

    for (index=0; index<samples; index++) {

        /* Read ADC input value */
        returnValue =
            mpiAdcInput(adc[index%adcCount],
                        (unsigned long*)&input);
        msgCHECK(returnValue);

        /* Record input information */
        sumInput+=input;
        sumInputSq+=input * 1.0 * input;

        if (index%adcCount == 0) {
            /* Wait one controller sample */
            meiControlSampleWait(mpiAdcControl(adc[0]),
                                 1);
        }
    }

    /* Calculate stastical input information */
    *inputMean = sumInput / samples;
    *inputVar = (sumInputSq - (sumInput*sumInput)/samples)/(samples-1);
}


/* Display ADC analysis */
void displayAdcAnalysis(double *lineVoltage,
                        double *lineUncertainty,
                        double *inputMean,
                        double *inputVar)
{
    double b;
    double bStdDev;
    double m;
    double mStdDev;
    double deltaInput;
    double deltaInputSq;


    deltaInput = inputMean[1] - inputMean[0];
    deltaInputSq = deltaInput * deltaInput;

    b = (lineVoltage[0]*inputMean[1] - lineVoltage[1]*inputMean[0]) /
            deltaInput;
```

```
    m = (lineVoltage[1] - lineVoltage[0]) / deltaInput;

    bStdDev = sqrt(
        ((b - lineVoltage[1])*(b - lineVoltage[1])*inputVar[0] +
            (b - lineVoltage[0])*(b - lineVoltage[0])*inputVar[1] +
            inputMean[0]*inputMean[0]*lineUncertainty[1]*lineUncertainty[1] +
            inputMean[1]*inputMean[1]*lineUncertainty[0]*lineUncertainty[0]) /
        deltaInputSq);

    mStdDev = sqrt(
        (m*m*(inputVar[0] + inputVar[1]) +
            lineUncertainty[0]*lineUncertainty[0] +
            lineUncertainty[1]*lineUncertainty[1]) /
        deltaInputSq);

    printf("Voltage = m * (ADC input) + b\n\n"
           " m = %le +/- %le \t(Volts/ADC_UNIT)\n"
           " b = %le +/- %le \t(Volts)\n\n",
           m, mStdDev,
           b, bStdDev);
}


int main(int     argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIAdc              adc[ADC_COUNT];

    double  inputMean[2];
    double  inputVar[2];
    double  lineVoltage[2]  =
        {LINE_VOLTAGE_1, LINE_VOLTAGE_2};
    double  lineUncertainty[2] =
        {LINE_VOLTAGE_UNCERTAINTY_1, LINE_VOLTAGE_UNCERTAINTY_2};

    long    index;

    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                adc,
                ADC_COUNT);

    /* Enable adcs */
    enableAdcs(control,
```

```
                ADC_COUNT);

    /* Configure adc objects to read LINE_INPUT_1 */
    for (index = 0; index < ADC_COUNT; index++) {
        adcConfigure(adc[index],
                     LINE_INPUT_1,
                     ADC_RANGE);
    }

    /* Delay so that adc object can read values from  LINE_INPUT_1 */
    meiPlatformSleep(1);

    /* Calculate stastical input information for LINE_INPUT_1 */
    getAdcInputInfo(adc,
                    ADC_COUNT,
                    SAMPLES,
                    &inputMean[0],
                    &inputVar[0]);

    /* Configure adc objects to read LINE_INPUT_2 */
    for (index = 0; index < ADC_COUNT; index++) {
        adcConfigure(adc[index],
                     LINE_INPUT_2,
                     ADC_RANGE);
    }

    /* Delay so that adc object can read values from  LINE_INPUT_1 */
    meiPlatformSleep(1);

    /* Calculate stastical input information for LINE_INPUT_2 */
    getAdcInputInfo(adc,
                    ADC_COUNT,
                    SAMPLES,
                    &inputMean[1],
                    &inputVar[1]);

    /* Display ADC analysis */
    displayAdcAnalysis(lineVoltage,
                       lineUncertainty,
                       inputMean,
                       inputVar);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   adc,
                   ADC_COUNT);

        return (MPIMessageOK);
}
```

adcFBak.c -- Set up an axis to use analog (ADC) position feedback

```
/* adcfbak.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Set up an axis to use analog (ADC) position feedback

  Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    adcfbakMain

argMainRENAME(main, adcfbak)
#endif

/* Command line arguments and defaults */
long    adcNumber   = 0;

Arg argList[] = {
    {   "-adc", ArgTypeLONG,    &adcNumber, },

    {   NULL,   ArgTypeINVALID, NULL,   }
};

#define ADC_COUNT           (1)     /* Configure controller to read 'n' ADCs */
#define ADC_BUFFER_SIZE     (100000)
#define ADC_RANGE           (10.0)  /* 10.0, 5.0, 2.5, or 1.25 volts */

#define AXIS_NUMBER         (0)

long
    axisAnalogFeedbackSet(MPIAxis   axis,
                          long      adcNumber);
```

```
int
    main(int    argc,
         char   *argv[])
{

    long    returnValue;
    long    argIndex;

    MPIControl          control;        /* Motion controller handle */
    MPIControlConfig    controlConfig;  /* Controller configuration */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIAxis             axis;
    MPIAdc              adc;
    MPIAdcConfig        adcConfig;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (adcNumber >= MEIXmpMAX_ADCs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-adc # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_ADCs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));
```

```
    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object*/
    axis =
        mpiAxisCreate(control,
                      AXIS_NUMBER);     /* axis number */
    msgCHECK(mpiAxisValidate(axis));

    /* Set the number of ADCs */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    controlConfig.adcCount = ADC_COUNT;     /* Enable 'n' ADCs */

    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);


    /* Create Adc object */
    adc =
        mpiAdcCreate(control,
                     adcNumber);
    msgCHECK(mpiAdcValidate(adc));

    /* Configure the ADC */
    returnValue =
        mpiAdcConfigGet(adc,
                        &adcConfig,
                        NULL);
    msgCHECK(returnValue);

    adcConfig.range = ADC_RANGE;    /* Voltage range */

    printf("\nADC Voltage Range: +/- %3.1lf volts\n",
            adcConfig.range);

    returnValue =
        mpiAdcConfigSet(adc,
                        &adcConfig,
                        NULL);
    msgCHECK(returnValue);

    returnValue =
        axisAnalogFeedbackSet(axis,
                              adcNumber);
```

```
    /* Delete the Adc handle */
    returnValue = mpiAdcDelete(adc);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    /* Delete the Control handle */
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return (returnValue);
}


long
    axisAnalogFeedbackSet(MPIAxis   axis,
                          long  adcNumber)
{
    MPIControl      control;
    MEIXmpData      *firmware;
    MPIAxisConfig   axisConfig;         /* axis configuration MPI */
    MEIAxisConfig   axisConfigXmp;      /* axis configuration XMP */

    long        returnValue;

    control = mpiAxisControl(axis);
    msgCHECK(mpiControlValidate(control));

    /* Get pointer to XMP firmware */
    returnValue =
        mpiControlMemory(control,
                         &firmware,
                         NULL);

    /* Configure axes: Gantry front end */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);

    axisConfigXmp.APos[0].Ptr    = &firmware->ADC[adcNumber].Input;

    /* Set axis configuration */
    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);
    return (returnValue);
}
```

ampFlt1.c -- Configure the amp fault input.

```
/* ampflt1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/ampflt1.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Configure the amp fault input.

The program will configure the amp fault input for a motor.

  A motor's amp fault input has four items to configure:

  1) Event Action   (any MPIAction such as MPIActionE_STOP)
  2) Event Trigger  (a trigger polarity, active HIGH or LOW)
  3) Direction Flag (TRUE will cause the command direction of motion to
                     qualify the events,    FALSE will ignore direction,
                     based solely on the limit input state)
  4) Duration       (requires the limit condition to exist for
                     a programmable number of seconds before an
                     event will occur)

The duration configuration provides filtering of the amp fault input by requiring the
input to
remain active for the defined duration.  This will effectively keep the amp fault
from activating prematurely
due to electrical noise on the amp fault input.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#define  ACTIVE_HIGH    (1)
#define  ACTIVE_LOW     (0)
```

```
#if defined(ARG_MAIN_RENAME)
#define main    ampflt1Main

argMainRENAME(main, ampflt1)
#endif

/* Command line arguments and defaults */
long            axisNumber      = 0;
long            motionNumber    = 0;
long            motorNumber     = 0;
float           ampfaultduration= (float)5.0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },
    {   "-duration",ArgTypeFLOAT,   &ampfaultduration,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;    /* motion controller handle */
    MPIAxis         axis;       /* axis handle(s) */
    MPIMotor        motor;      /* motor handle(s) */
    MPIMotion       motion;     /* motion supervisor handle */
    MPINotify       notify;     /* event notification handle */
    MPIEventMgr     eventMgr;   /* event manager handle */

    Service service;

    MEIXmpData  *firmware;

    MPIMotorEventConfig eventConfig;

    long    messageCount;

    long    returnValue;    /* return value from library */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIEventMask        eventMask;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;
```

```
        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motorNumber  >= MEIXmpMAX_Motors) ||
        (ampfaultduration  < 0)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n"
                           "\t\t[-duration # (>0)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Obtain a Control handle. */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Get pointer to XMP firmware */
    returnValue =
        mpiControlMemory(control,
                         (void**)&firmware,
                         (void**)NULL);
    msgCHECK(returnValue);

    /* Create axis object using axisNumber on controller */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motor object using axisNumber on controller */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));
```

```
    /* Create motion supervisor object using MS number */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Get configuration for ampfault event */
    returnValue =
        mpiMotorEventConfigGet(motor,
                               MPIEventTypeAMP_FAULT,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Event Action, configure to ABORT on Amp Fault Input */
    eventConfig.action = MPIActionABORT;

    /* Event trigger, set trigger level to Active Low */
    eventConfig.trigger.polarity = ACTIVE_LOW;

    /* Direction Flag, do not use commanded velocity direction as a trigger qualifier
 */
    eventConfig.direction = FALSE;
```

```
    /* Duration, wait ampfaultduration seconds before triggering */
    eventConfig.duration = ampfaultduration;

    returnValue =
        mpiMotorEventConfigSet(motor,
                               MPIEventTypeAMP_FAULT,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    meiPlatformConsole("amp fault duration and polarity configured: wait for
events.\n");

    messageCount = 0;

    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus  eventStatus;

        long    motorNumber;

        /* Wait for event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);
        msgCHECK(returnValue);

        messageCount++;

        motorNumber = ((MEIEventStatusInfo *)eventStatus.info)->type.number;

        printf("Event! (%d) Type = %d \n",
               messageCount,
               eventStatus.type);

        switch (eventStatus.type) {
            case MPIEventTypeAMP_FAULT: {


                printf("Amp Fault has occurred on motor # %d\n",motorNumber);
                printf("Resetting...\n");

                returnValue =
                    mpiMotionAction(motion,
                                    MPIActionRESET);
                meiASSERT(returnValue == MPIMessageOK);

                break;
            }
            default: {
                break;
            }
        }
    }

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
```

```
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

anticol1.c -- Program Sequencer aborts axes prior to a collision (anti-collision protection)

```
/* anticol1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/Anticol1.c 6     7/23/01 2:36p Kevinh $";
#endif


/*

:Program Sequencer aborts axes prior to a collision (anti-collision protection)

This sample program provides anti-collision protection for two axes that move
 along the same physical space.  The program creates a program sequencer that
 will continually monitor the relative actual positions of two axes.  When the
 relative positions get closer than a defined tolerance (defined at the top of
 the program as DIFFERENCE_TOLEARNCE) the axes will be aborted by the program
 sequencer.

The XMP program sequencer controls the execution of a single command or a
 series of commands on the XMP controller.  The program sequencer provides the
 ability to execute programs directly on the XMP controller without host
 intervention.  Examples of individual commands that can be executed by the
 program sequencer are motion, looping, conditional branching, computation,
 reading and writing of memory, time delays, waiting for conditions, setting
 inputs/outputs, and generation of events.  This rich command set provides
 capability similar to, and beyond, that provided by Programmable Logic
 Controller (PLC) programs.

From anticol1c.c (8/1/1999)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#if defined(ARG_MAIN_RENAME)
#define main    anticol1Main

argMainRENAME(main, anticol1)
#endif

#define AXIS_COUNT              (2)
#define DIFFERENCE_TOLERANCE    (7000)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
long            sequenceNumber  = 0;
long            differenceTolerance = DIFFERENCE_TOLERANCE;

Arg argList[] = {
    {   "-axis",        ArgTypeLONG,    &axisNumber[0],         },
    {   "-motion",      ArgTypeLONG,    &motionNumber,          },
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,        },
    {   "-tolerance",   ArgTypeLONG,    &differenceTolerance,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};


                        /* COMPUTE + BRANCH + BRANCH + ABORT = 4 */
MPICommand  CommandTable[4];
#define COMMAND_COUNT   (sizeof(CommandTable) / sizeof(MPICommand))

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIMotion   motion;             /* motion handle */
    MPIAxis     axis[AXIS_COUNT];   /* axis handles */
    MPISequence sequence;           /* sequence handle */

    MPICommandParams    commandParams;  /* command parameters */

    long    returnValue;    /* return value from library */

    long    commandIndex;   /* CommandTable[] index */
    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MEIXmpData          *firmware;
    MEIPlatform         platform;   /* platform handle */
    MEIXmpBufferData    *xmpBufferData;
    long                *addressActPos1;
    long                *addressActPos2;
    long                address;
    long                *allocPointer;
    long                difference;
    long                *addressDifferenceStored;
```

```
    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]  > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber   >= MEIXmpMAX_MSs) ||
        (sequenceNumber >= MEIXmpMAX_PSs) ||
        (differenceTolerance < 0)){

        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[-tolerance # ( larger than 0)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_PSs - 1);

        exit(MPIMessageARG_INVALID);
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    if (returnValue != MPIMessageOK) {
```

```
        fprintf(stderr, "%s: mpiControlInit() returns 0x%x: %s\n",
                        argv[0],
                        returnValue,
                        mpiMessage(returnValue, NULL));
        exit(1);
    }

    /* Get pointer to XMP firmware */
    returnValue =
        mpiControlMemory(control,
                         &firmware,
                         &xmpBufferData);
    msgCHECK(returnValue);

    platform =
        meiControlPlatform(control);
    msgCHECK(meiPlatformValidate(platform));

    /*Pointer into xmp memory space.  Pointer to the user buffer  */
    returnValue =
        mpiControlMemoryAlloc(control,
                              MPIControlMemoryTypeUSER,
                              sizeof(long),
                              &allocPointer);
    msgCHECK(returnValue);

    returnValue =
            meiPlatformMemoryToFirmware(platform,
                                        allocPointer,
                                        (void **)&address);
    msgCHECK(returnValue);

    meiPlatformConsole("address of allocPointer (user buffer) is 0x%x\n",address);

    /* Create motion object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create axis object for axis #0 */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);   /* axis #0 */
    msgCHECK(mpiAxisValidate(axis[0]));

    /* Create axis object for axis #1 */
    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);   /* axis #1 */
    msgCHECK(mpiAxisValidate(axis[1]));

    /* Create motion axis list */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
```

```
                            axis);
    msgCHECK(returnValue);

    /* Write the axis list to the motion supervisor on the board */
    returnValue =
        mpiMotionAction(motion,
                        MEIActionMAP);

    msgCHECK(returnValue);

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          -1);
    msgCHECK(mpiSequenceValidate(sequence));

    /* CommandTable[commandIndex] */
    commandIndex = 0;

    /* Get the address for the first axis's actual position register */
    addressActPos1 = &firmware->Axis[axisNumber[0]].ActualPosition;

    /* Get the address for the second axis's actual position register */
    addressActPos2 = &firmware->Axis[axisNumber[1]].ActualPosition;

    returnValue =
            meiPlatformMemoryToFirmware(platform,
                                        addressActPos1,
                                        (void **)&address);
    msgCHECK(returnValue);

    meiPlatformConsole("addressActPos1 is 0x%x\n",address);

    returnValue =
            meiPlatformMemoryToFirmware(platform,
                                        addressActPos2,
                                        (void **)&address);
    msgCHECK(returnValue);
    meiPlatformConsole("addressActPos2 is 0x%x\n",address);

    /* 1st. Command  */
    /* Take the difference of the two actual positions and store them in the user
 buffer */
    addressDifferenceStored = allocPointer;

    commandParams.compute.dst.l         = addressDifferenceStored;
    commandParams.compute.expr.address.l    = addressActPos1;
    commandParams.compute.expr.oper     = MPICommandOperatorSUBTRACT;
    commandParams.compute.expr.by.ref.l = addressActPos2;

    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeCOMPUTE_REF,
                         &commandParams,
                         "First");
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));
```

```
    commandIndex++;

    /*  2nd Command */
    /*  Compare the position differences to the position diference tolerance */
    /*  Perform abort if test is true */

    commandParams.branch.label              = "ABORT";
    commandParams.branch.expr.address.l     = addressDifferenceStored;
    commandParams.branch.expr.oper          = MPICommandOperatorLESS;
    commandParams.branch.expr.by.value.l    = differenceTolerance;

    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* 3rd. Command */
    /* Branch to the first command of the sequence */
    commandParams.branch.label      = "First";  /* First command */
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* 4th. command */
    /* Abort the axes that are part of motion supervisor motion */

    commandParams.motion.motionCommand  = MPICommandMotionABORT;
    commandParams.motion.motion     = motion;

    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeMOTION,
                         &commandParams,
                         "ABORT");
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;


    /* Create sequence command list */
    returnValue =
        mpiSequenceCommandListSet(sequence,
                                  commandIndex,
                                  CommandTable);
    msgCHECK(returnValue);

    /* Start sequence */
    returnValue =
```

```
        mpiSequenceStart(sequence,
                         MPIHandleVOID);

    if (returnValue != MPIMessageOK) {
        fprintf(stderr, "%s: mpiSequenceStart() returns 0x%x: %s\n",
                        argv[0],
                        returnValue,
                        mpiMessage(returnValue, NULL));
        exit(2);
    }

    meiPlatformConsole("Press any key to stop sequence\n");
    meiPlatformKey(MPIWaitFOREVER);

    returnValue =
            mpiControlMemoryGet(control,
                                &difference,
                                allocPointer,
                                sizeof(long));
    msgCHECK(returnValue);

    meiPlatformConsole("Axis Position difference is %ld\n",difference);

    /* Stop sequence */
    returnValue = mpiSequenceStop(sequence);
    msgCHECK(returnValue);

    returnValue = mpiSequenceDelete(sequence);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (index = 0; index < AXIS_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue =
        mpiControlMemoryFree(control,
                             MPIControlMemoryTypeUSER,
                             sizeof(long),
                             allocPointer);
    msgCHECK(returnValue);


    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

capture1.c -- Perform a velocity move and capture position based on Home or Index pulse.

```
/* capture1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/capture1.c 10    7/18/01 9:32a Kevinh $"
#endif

#if defined(ARG_MAIN_RENAME)
#define main    capture1Main

argMainRENAME(main, capture1)
#endif

/*

:Perform a velocity move and capture position based on Home or Index pulse.

This sample application demonstrates use of the Capture feature on the XMP.
First, a velocity move is commanded on an Axis.  A capture is then configured
to trigger on either a Home sensor or index pulse.

Each motion block supports 10 capture registers.   The default configuration
 is two capture registers per motor -- while the last two (8,9) on each motion
 block are reserved for the Auxiliary Encoder (not supported).   The capture
 registers are default mapped as follows... 0 & 1 for Motor0, 2 & 3 for Motor1,
 10 & 11 for Motor4, etc.  The first Capture for each motor latches the default
 (primary) encoder input, and the second capture latches the AUX encoder
 input. The equation used below calculates the Capture number for the primary
 motor feedback given the default capture mapping.

    captureNumber = ((motorNumber / MEIXmpMotorsPerBlock) * MEIXmpMaxLatches) +
                    ((motorNumber % MEIXmpMotorsPerBlock) * CapturesPerMotor);

This program presumes the tuning parameters(PID, PIV, etc.) for your
stage(or tool, or machine) have already been set.  That is, the tuning
parameters need to be established prior to running this program so the
stage can move in a stable manner.  Please use the Tuning Guidelines if
these values haven't been determined.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
```

```
*/

#include <stdlib.h>
#include <stdio.h>
#include "stdmpi.h"
#include "stdmei.h"
#include "apputil.h"


#define CapturesPerMotor    (2) /* Default Capture configuration */

/* User Settings */
#define ACTIVE_FALLING_EDGE     (0)
#define ACTIVE_RAISING_EDGE     (1)

#define MOTORNUM            (0) /* Number of motors */

/* Motion Parameters */
#define VELOCITY    (5000.0)    /* Move velocity */
#define ACCEL       (10000.0)   /* Move acceleration*/

/* Capture Parameters */
#define CAPTURE_EDGE        (ACTIVE_FALLING_EDGE) /* Capture on falling edge */

/* CAPTURE_TRIGGER can be MEIMotorInputHOME or MEIMotorInputINDEX */
#define CAPTURE_TRIGGER     (MEIMotorInputHOME)   /* Capture on home pulse */


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                   argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;


    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr, "usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Calculate default capture number for axisNumber */
long captureNumber(long     motorNumber)
{
    return  ((motorNumber/MEIXmpMotorsPerBlock) * MEIXmpMaxLatches) +
            ((motorNumber % MEIXmpMotorsPerBlock) * CapturesPerMotor);
}
```

```
/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber,
                 MPIMotor           *motor,
                 long                motorNumber,
                 MPICapture         *capture,
                 long                captureNumber)
{
    long    returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        *axis);
    msgCHECK(mpiMotionValidate(*motion));

    /* Create motor object */
    *motor =
        mpiMotorCreate(*control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(*motor));

    /* Create capture object */
    *capture =
        mpiCaptureCreate(*control,
                         captureNumber);
    msgCHECK(mpiCaptureValidate(*capture));
}
```

```
/*
   Configure capture object.

   edge: 0 for falling edge, 1 for rising edge
*/
void configureCapture(MPICapture    capture,
                      MEIMotorInput trigger,
                      long          edge)
{
    MPICaptureConfig    captureConfig;
    long                returnValue;


    /* Disable capture */
    returnValue =
        mpiCaptureArm(capture,
                      FALSE);
    msgCHECK(returnValue);

    /* Read caputre configuration */
    returnValue =
        mpiCaptureConfigGet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);

    /* Set capture parameters */
    captureConfig.trigger.mask    = trigger;
    captureConfig.trigger.pattern = edge ? trigger : 0;

    /* Write capture configuration */
    returnValue =
        mpiCaptureConfigSet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);
}


/* Disable Home Event action */
void disableHomeEvent(MPIMotor      motor,
                      MPICapture    capture)
{
    MPIMotorEventConfig eventConfig;
    long                returnValue;


    returnValue =
        mpiMotorEventConfigGet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;     /* No action   */
```

```
    eventConfig.trigger.polarity = TRUE;    /* Active high */

    returnValue =
        mpiMotorEventConfigSet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Arm the capture */
    returnValue =
        mpiCaptureArm(capture,
                      TRUE);
    msgCHECK(returnValue);
}


/* Perform a Velocity move */
void velocityMove(MPITrajectory     *trajectory,
                  MPIMotionParams   *params,
                  MPIMotion         *motion)
{
    long    returnValue;

    /* Setup motion parameters */
    trajectory->velocity = VELOCITY;
    trajectory->acceleration = ACCEL;
    trajectory->deceleration = ACCEL; /* Not used for velocity move */
    trajectory->jerkPercent = 0.0;    /* Not used for velocity move */
    params->velocity.trajectory = trajectory;

    /* Start a velocity move */
    returnValue =
        mpiMotionStart(*motion,
                       MPIMotionTypeVELOCITY,
                       params);
    msgCHECK(returnValue);
}


/* Poll capture status and update display */
void pollForCaptureAndUpdateDisplay(MPICapture      *capture,
                                    MPIMotor        *motor,
                                    MEIMotorInput    trigger)
{
    MPICaptureStatus    captureStatus;
    MPIMotorIo          io;
    long                caputuresCounted = 0;
    long                returnValue;


    /* State Machine - Poll capture status, update display, etc. */
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        returnValue =
            mpiCaptureStatus(*capture,
```

```
                                &captureStatus,
                                NULL);
        msgCHECK(returnValue);

        /* Display Home and Capture state */
        returnValue =
            mpiMotorIoGet(*motor,
                          &io);
        msgCHECK(returnValue);

        printf("\rI/O state:%d  CaptureState:%d  ",
               !(!(io.input & trigger)),    /* input bit */
               captureStatus.state);        /* (1=ARMED, 2=CAPTURED) */

        if (captureStatus.state == MPICaptureStateCAPTURED) {
            caputuresCounted++;

            printf("Latched Position = %.0lf  # of captures = %d\n\n",
                   captureStatus.latch[0],
                   caputuresCounted);

            /* Re-arm position capture */
            returnValue =
                mpiCaptureArm(*capture,
                              TRUE);
            msgCHECK(returnValue);
        }
    }
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis,
                    MPIMotor        *motor,
                    MPICapture      *capture)
{
    long    returnValue;


    /* Delete capture object */
    returnValue =
        mpiCaptureDelete(*capture);
    msgCHECK(returnValue);

    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);

    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
    msgCHECK(returnValue);
```

```
    /* Delete axis object */
    returnValue =
        mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


int main(int    argc,
         char   *argv[])
{

    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;
    MPIAxis             axis;
    MPIMotor            motor;
    MPICapture          capture;
    MPITrajectory       trajectory;
    MPIMotionParams     params;

    long    returnValue;                /* Return value from library */
    long    motorNumber = MOTORNUM;
    long    motionNumber = motorNumber;
    long    axisNumber = motorNumber;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                motionNumber,
                &axis,
                axisNumber,
                &motor,
                motorNumber,
                &capture,
                captureNumber(motorNumber));

    /* Configure capture  */
    configureCapture(capture,
                     CAPTURE_TRIGGER,
```

```
                    CAPTURE_EDGE);

    /* Disable Home Event action */
    disableHomeEvent(motor,
                    capture);

    /* Perform a Velocity move */
    velocityMove(&trajectory,
                 &params,
                 &motion);

    printf("\n Moving.... Press any key to quit.\n\n");

    /* Poll capture status and update display */
    pollForCaptureAndUpdateDisplay(&capture,
                                   &motor,
                                   CAPTURE_TRIGGER);

    printf("\nStopping.\n");

    /* Stop motion on axis */
    returnValue =
        mpiMotionAction(motion,
                        MPIActionSTOP);
    msgCHECK(returnValue);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis,
                   &motor,
                   &capture);

    return ((int)returnValue);
}
```

coffee.c -- Read XMP firmware signature

```
/* coffee.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/coffee.c 10    7/23/01 2:36p Kevinh $";
#endif

/*

:Read XMP firmware signature

This sample application reads the firmware signature and other controller
ID registers including:  Signature, Disable, SoftwareID, and Option.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    coffeeMain

argMainRENAME(main, coffee)
#endif

unsigned long MEIMemory[64];

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* Motion controller handle */

    MPIControlType      controlType;
```

```
    MPIControlAddress   controlAddress;

    void    *memory;        /* Control memory */

    long    returnValue;    /* Return value from library */
    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n",
                           argv[0],
                           ArgUSAGE);
        exit(MPIMessagePARAM_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    returnValue =
        mpiControlMemory(control,
                         &memory,
                         NULL);
    msgCHECK(returnValue);

    /* Read and Print the Controller firmware signature */
    returnValue =
        mpiControlMemoryGet(control,
                            MEIMemory,
                            memory,
                            sizeof(MEIMemory));
    msgCHECK(returnValue);

    fprintf(stderr, "Sharc memory: 0x0%08x 0x%08x 0x%08x 0x%08x\n",
                    MEIMemory[0],
                    MEIMemory[1],
                    MEIMemory[2],
                    MEIMemory[3]);

    /* Delete objects */
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);
```

```
    return ((int)returnValue);
}
```

comp.c -- Configure Axis Compensation Tables

```
/* comp.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/comp.c 10    7/23/01 2:36p Kevinh $";
#endif

/*

:Configure Axis Compensation Tables

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    compMain

argMainRENAME(main, comp)
#endif

#define COMP_FILENAME   "comp.txt"

MEIXmpCompensator   Comp;

long    CompTable[MEIXmpCompTableSize];

long    loadCompTable(MPIControl        control,
                      long              index,
                      MEIXmpCompensator *comp);

int
    main(int    argc,
         char   *argv[])
{
```

```
    MPIControl  control;    /* motion controller handle */

    char    *fileName;
    FILE    *fp;
    long    compIndex;
    long    compIndexOld;
    long    controlIndex;
    long    tableIndex;
    long    tableCount = 0;
    char    buffer[256];

    long    returnValue;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    fileName =
        (argIndex >= argc)
                ? COMP_FILENAME
                : argv[argIndex++];

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[fileName (%s)]",
                           argv[0],
                           ArgUSAGE,
                           COMP_FILENAME);
        exit(0);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    fp = fopen(fileName, "r");

    if (fp == NULL) {
        fprintf(stderr,
                "%s: file not found.\n",
                fileName);
        exit(2);
    }
```

```
    compIndex = 0;
    compIndexOld = -1;

    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        long    inAxis;
        long    outAxis;
        long    minPosition;
        long    deltaPosition;
        long    compPoints;
        long    dimension;

        char    *ptr;

        if (compIndex > compIndexOld) {
            Comp.Dimension                  = -1;
            Comp.OutAxis                    = -1;
            Comp.Control[0].InAxis          = -1;
            Comp.Control[0].DeltaPosition   = -1;
            Comp.Control[0].CompPoints      = -1;
            Comp.Control[1].InAxis          = -1;
            Comp.Control[1].DeltaPosition   = -1;
            Comp.Control[1].CompPoints      = -1;
            tableCount = 0;


            memset(CompTable, 0, sizeof(CompTable));
            Comp.CompTable = CompTable;

            compIndexOld = compIndex;
        }

        for (ptr = buffer; *ptr != '\0'; ptr++) {
            char    byte = *ptr;

            if (islower(byte)) {
                *ptr = (char)toupper(byte);
            }
        }

        if (sscanf(buffer, "#DIMENSION  = %d", &dimension) == 1) {
            Comp.Dimension = dimension;
            continue;
        }

        if (sscanf(buffer, "#INAXIS[%d]  = %d",&controlIndex, &inAxis) == 2) {
            Comp.Control[controlIndex].InAxis = inAxis;
            continue;
        }

        if (sscanf(buffer, "#OUTAXIS  = %d", &outAxis) == 1) {
            Comp.OutAxis = outAxis;
            continue;
        }

        if (sscanf(buffer, "#MINPOSITION[%d]  = %d",&controlIndex, &minPosition) ==
 2) {
            Comp.Control[controlIndex].MinPosition = minPosition;
            continue;
```

```
        }

        if (sscanf(buffer, "#DELTAPOSITION[%d]  = %d",&controlIndex, &deltaPosition)
== 2) {
            Comp.Control[controlIndex].DeltaPosition = deltaPosition;
            continue;
        }

        if (sscanf(buffer, "#COMPPOINTS[%d]  = %d",&controlIndex, &compPoints) == 2)
{
            Comp.Control[controlIndex].CompPoints = compPoints;
            continue;
        }

        if (sscanf(buffer, "#POSITION[%d]", &tableIndex) == 1) {
            long    index;
            long    calc_x;

            if (Comp.Dimension == 1) {
                Comp.Control[1].CompPoints = 1;
                Comp.Control[1].InAxis          = 0;
                Comp.Control[1].DeltaPosition   = 0;
            }

            /* Do some error checking. */
            if (compIndex >= MEIXmpMAX_Compensators) {
                fprintf(stderr,
                        "Too many compensators. Max = %d.\n",
                        MEIXmpMAX_Compensators);
                break;
            }

            if (Comp.OutAxis < 0) {
                fprintf(stderr,
                        "No #OutAxis statement.\n");
                break;
            }

            if (Comp.Dimension < 0) {
                fprintf(stderr,
                        "No #Dimension statement.\n");
                break;
            }

            for (controlIndex = 0; controlIndex < Comp.Dimension; controlIndex++) {
                if (Comp.Control[controlIndex].InAxis < 0) {
                    fprintf(stderr,
                            "No #InAxis[%d] statement.\n",
                            controlIndex);
                    break;
                }

                if (Comp.Control[controlIndex].CompPoints < 0) {
                    fprintf(stderr,
                            "No #CompPoints[%d] statement.\n",
                            controlIndex);
                    break;
                }
```

```
            if (Comp.Control[controlIndex].DeltaPosition < 0) {
                fprintf(stderr,
                        "Delta Position[%d] (%d) is invalid.\n",
                        controlIndex,
                        Comp.Control[controlIndex].DeltaPosition);
                break;
            }
        }
        if ((Comp.Control[0].CompPoints * Comp.Control[1].CompPoints)
                > MEIXmpCompTableSize) {
            fprintf(stderr,
                    "Too many compensation points. Max = %d.\n",
                    MEIXmpCompTableSize);
            break;
        }

        if (tableIndex >= Comp.Control[1].CompPoints) {
            fprintf(stderr,
                    "Compensation table number (%d) out of range [0-%d].\n",
                    tableIndex,
                    Comp.Control[1].CompPoints);
            break;
        }

        calc_x = Comp.Control[0].MinPosition;

        index = 0;

        while (index < Comp.Control[0].CompPoints) {
            long    x;
            long    y;
            long    tableOffset;

            if (fgets(buffer, sizeof(buffer), fp) == NULL) {
                fprintf(stderr,
                        "Unexpected end-of-file\n");
                break;
            }

            if (sscanf(buffer, "%d %d", &x, &y) == 2) {
                if (x != calc_x) {
                    fprintf(stderr,
                            "Table out of order, expected position = %d, read
 %d\n",
                            calc_x,
                            x);
                    break;
                }
                tableOffset = index + tableIndex * Comp.Control[0].CompPoints;
                Comp.CompTable[tableOffset] = y;
                calc_x += Comp.Control[0].DeltaPosition;
                index++;
            }
        }

        if (index < Comp.Control[0].CompPoints) {
            fprintf(stderr,
```

```
                        "Compensation table number (%d) too small.\n",
                        tableIndex);
                break;
            }
            else {
                tableCount++;
                fprintf(stderr,
                        "Compensation Table %d read.\n",
                        tableIndex);
            }

            if (tableCount >= Comp.Control[1].CompPoints) {
                returnValue =
                    loadCompTable(control,
                                  compIndex,
                                  &Comp);

                fprintf(stderr,
                        "Compensator %d is %sconfigured.\n",
                        compIndex,
                        (returnValue == MPIMessageOK)
                                    ? ""
                                    : "NOT ");

                compIndex++;
            }
        }
    }

    fclose(fp);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long loadCompTable(MPIControl           control,
                   long                 index,
                   MEIXmpCompensator    *comp)
{
    MPIControlConfig    controlConfig;
    MEIControlConfig    controlConfigXmp;

    long    returnValue;

    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            &controlConfigXmp);

    if (returnValue == MPIMessageOK) {
        if (controlConfigXmp.Compensator[index].CompTable != NULL) {

            fprintf(stderr,
                    "Warning: controlConfigXmp.Compensator[%d] is already in use\n",
                    index);
        }
```

```
        controlConfigXmp.compensatorCount = index + 1;
        controlConfigXmp.Compensator[index] = *comp;

        if (index > 0) {
            long *prev_dest;
            long prev_size;

            prev_dest = controlConfigXmp.Compensator[index - 1].CompTable;
            prev_size = controlConfigXmp.Compensator[index - 1].Control[0].CompPoints
*
               controlConfigXmp.Compensator[index-1].Control[1].CompPoints;

            controlConfigXmp.Compensator[index].CompTable = &prev_dest[prev_size];
        }
        else {
            controlConfigXmp.Compensator[index].CompTable =
                controlConfigXmp.CompensationTable;
        }

        meiASSERT(&controlConfigXmp.CompensationTable[MEIXmpCompTableSize] >=
                  &controlConfigXmp.Compensator[index].CompTable[
                            controlConfigXmp.Compensator[index].Control[0].CompPoints
*

controlConfigXmp.Compensator[index].Control[1].CompPoints]);

        memcpy(controlConfigXmp.Compensator[index].CompTable,
               comp->CompTable,
               controlConfigXmp.Compensator[index].Control[0].CompPoints *
               controlConfigXmp.Compensator[index].Control[1].CompPoints *
                    sizeof(long));

        returnValue =
            mpiControlConfigSet(control,
                                &controlConfig,
                                &controlConfigXmp);
    }

    return (returnValue);
}
```

compare1.c -- Compare an encoder counter and set an XCVR output when the condition is TRUE.

```
/* compare1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/compare1.c 7     7/23/01 2:36p Kevinh $"
#endif

/*

:Compare an encoder counter and set an XCVR output when the condition is TRUE.

Each motion block supports 10 compare registers.   The default configuration
 is two compare registers per motor -- while the last two (8,9) on each motion
 block are reserved for the Auxiliary Encoder (not supported).   The compare
 registers are default mapped as follows.... 0 & 1 for Motor0, 2 & 3 for Motor1,
 10 & 11 for Motor4, etc.  The first Compare for each motor uses the default
 (primary) encoder input for position compare, the second uses the AUX encoder
 input. The equation used below calculates the Compare number for the primary
 motor feedback given the default compare mapping.

    compareNumber = ((motorNumber / MEIXmpMotorsPerBlock) *
MEIXmpMaxComparePositions) +
                    ((motorNumber % MEIXmpMotorsPerBlock) * ComparesPerMotor);

A compare register is loaded with a position compare value, a compare operator
 (MPICommandOperatorGREATER or MPICommandOperatorLESS_OR_EQUAL), and an output
 state (sets the Motor's XCVR_C TRUE or FALSE when the compare state is true).

A compare's status can be polled, it does not generate an event.  If the
 compare register's state equals MPICompareStateCOMPARED, the compare
 position is stored and the compare register is ready for re-arming.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#if defined(ARG_MAIN_RENAME)
#define main    compare1Main

argMainRENAME(main, compare1)
#endif

#define MOTOR           (4)
#define COMPARE_POS     (15000)
#define ComparesPerMotor    (2) /* Default Compare configuration */

/* XCVR Settings */
#define TRANSCEIVER_CONFIG  (MEIMotorTransceiverConfigCOMPARE) /* XCVR will output
Compare logic */
#define TRANSCEIVER_ID      (MEIMotorTransceiverIdC)  /* Must be XCVR_C for Compare
*/
#define TRANSCEIVER_MASK    (MEIMotorTransceiverMaskC) /* same as above */


int
    main(int    argc,
         char   *argv[])
{
    MPIControl          control;        /* motion controller object */
    MPIAxis             axis;           /* axis object */
    MPIMotor            motor;          /* motor object */
    MEIMotorConfig      motorConfigXmp;  /* contains transceiver configuration */
    MPIMotorIo          io;
    MPICompare          compare;        /* compare object */

    MPICompareParams    compareParams;  /* compare parameters */
    MPICompareStatus    compareStatus;  /* compare status */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    returnValue;        /* return value from library */
    long    argIndex;
    long    motorNumber;
    long    compareNumber;
    double  origin;
    double  actualPosition;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    motorNumber =
        (argIndex >= argc)
                ? MOTOR
                : meiPlatformAtol(argv[argIndex++]);

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
```

```
                            "\t\t[motor# (0)]\n",
                             argv[0],
                             ArgUSAGE);
        exit(0);
    }

    /* Calculate default compare number for axisNumber */
    compareNumber = ((motorNumber/MEIXmpMotorsPerBlock) * MEIXmpMaxComparePositions)
+
                    ((motorNumber % MEIXmpMotorsPerBlock) * ComparesPerMotor);

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object */
    axis =
        mpiAxisCreate(control,
                      motorNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Read the Axis' Origin */
    returnValue =
        mpiAxisOriginGet(axis,
                         &origin);
    msgCHECK(returnValue);

    /* Create motor object for axisNumber */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));


    /* Configure selected transceiver */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Transceiver[TRANSCEIVER_ID].Config = TRANSCEIVER_CONFIG;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Create compare object for compareNumber */
```

```
    compare =
        mpiCompareCreate(control,
                         compareNumber);
    msgCHECK(mpiCompareValidate(compare));

    /* Disable compare */
    returnValue =
        mpiCompareArm(compare,
                      FALSE);
    msgCHECK(returnValue);

    /* Set compare parameters */

    /* remember to use the Axis' Origin when calculating the Compare position */
    compareParams.position = (long) (COMPARE_POS + origin);

    /* Configures state of the XCVR output, when Compare is valid */
    compareParams.outputState = TRUE;

    /*
        commandOperator must be MPICommandOperatorGREATER
        or MPICommandOperatorLESS_OR_EQUAL -- anything else is INVALID
     */
    compareParams.commandOperator = MPICommandOperatorGREATER;

    /* Load params */
    returnValue =
        mpiCompareLoad(compare,
                       &compareParams,
                       NULL);
    msgCHECK(returnValue);

    /* Arm the compare */
    returnValue =
        mpiCompareArm(compare,
                      TRUE);
    msgCHECK(returnValue);

    /* State Machine - Poll compare status, update display, etc */
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        returnValue =
            mpiCompareStatus(compare,
                             &compareStatus,
                             NULL);
        msgCHECK(returnValue);

        /* Display XCVR, Compare state and Axis' actual position */
        returnValue =
            mpiMotorIoGet(motor,
                          &io);
        msgCHECK(returnValue);

        returnValue =
            mpiAxisActualPositionGet(axis,
                                     &actualPosition);
        msgCHECK(returnValue);
```

```
        printf("\rXCVR_C:0x%x  State:0x%x  ActualPosition:%.0lf",
                io.input & TRANSCEIVER_MASK,    /* XCVR bit, read as input -- input
from FPGA */
                compareStatus.state,            /* value of Compare bit (1=ARMED,
2=COMPARED) */
                actualPosition);

        if (compareStatus.state == MPICompareStateCOMPARED) {
            printf("  Compared Position:%.0lf\r",
                    (double) (compareStatus.position[compareNumber %
MEIXmpMaxComparePositions] - origin));

            /* Re-arm position compare */
            returnValue =
                mpiCompareArm(compare,
                              TRUE);
            msgCHECK(returnValue);
        }
        else {
            /* clear any previous positions from the display... */
            printf("                                ");
        }
    }

    /* Disarm position compare */
    returnValue =
        mpiCompareArm(compare,
                      FALSE);
    msgCHECK(returnValue);

    /* Delete objects */

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiCompareDelete(compare);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

---

encfltcfg.c -- Configure encoder fault settings for a motor.

```
/* encfltcfg.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/encfltcfg.c 1     2/05/02 11:33a Erikb $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    limitsw1Main
argMainRENAME(main, limitsw1)
#endif

/*

:Configure encoder fault settings for a motor.

Encoder faults are set with an action, duration, and trigger.

  The action is one of the following:

        MPIActionSTOP,
        MPIActionE_STOP,
        MPIActionE_STOP_ABORT,
        MPIActionABORT,

The duration is in seconds. Keep in mind that the resolution of this setting
  is one servo sample period (1/sample rate).

The trigger sets three switches.
        MPIMotorEncoderFaultMaskBW_DET,         Broken wire switch
        MPIMotorEncoderFaultMaskILL_DET,        Illegal state switch
        MPIMotorEncoderFaultMaskABS_ERR,        Absolute encoder error switch

Broken wires occur when the positive and negative halves of an encoder signal
  pull together. An example would be when A+ and A- are at the same voltage
  because one of the A wires is disconnected.

Illegal state occurs when the A and B channel change level within 40 nsec.

Absolute encoder errors occur when there is a communication error on an
  absolute encoder. The details of this will vary with different types of
  absolute encoders.

Events are configured using MPIMotorEventConfigGet/Set(...).
```

```
 Warning!  This is a sample program to assist in the integration of the
  XMP motion controller with your application.  It may not contain all
  of the logic and safety features that your application requires.

 */

 #include <stdlib.h>
 #include <stdio.h>

 #include "stdmpi.h"
 #include "stdmei.h"

 #include "apputil.h"


 #define MOTOR           (0)
 #define EVENT_DURATION  (0.010)     /* seconds */


 /* Perform basic command line parsing. (-control -server -port -trace) */
 void basicParsing(int                    argc,
                   char                  *argv[],
                   MPIControlType        *controlType,
                   MPIControlAddress     *controlAddress)
 {
     long argIndex;

     /* Parse command line for Control type and address */
     argIndex = argControl(argc, argv, controlType, controlAddress);

     /* Check for unknown/invalid command line arguments */
     if (argIndex < argc) {
         fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
         exit(MPIMessageARG_INVALID);
     }
 }


 void encoderFaultConfigure(MPIMotor        motor,
                                            MPIAction       action,
                                            double          duration)
 {
     MPIMotorEventConfig eventConfig;
     long returnValue;

     /* Get the current limit configuration */
     returnValue =
         mpiMotorEventConfigGet(motor,
                                MPIEventTypeENCODER_FAULT,
                                &eventConfig,
                                NULL);
     msgCHECK(returnValue);

       /* Wait duration seconds before triggering */
     eventConfig.duration = (float) duration;

     /* Configure motor to perform action upon limit */
```

```
    eventConfig.action = action;

        /* Turn on all encoder fault types */
        eventConfig.trigger.mask =      MPIMotorEncoderFaultMaskBW_DET |
                MPIMotorEncoderFaultMaskILL_DET |
                MPIMotorEncoderFaultMaskABS_ERR;

    /* Set the new limit configuration */
    returnValue =
        mpiMotorEventConfigSet(motor,
                               MPIEventTypeENCODER_FAULT,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotor           *motor,
                 long                motorNumber)
{
    long returnValue;


    /* Obtain a Control handle */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize the controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Obtain a Motor handle */
    *motor =
        mpiMotorCreate(*control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(*motor));
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIMotor    *motor)
{
    long returnValue;

    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);
```

```
    /* Delete control object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;
    MPIMotor            motor;
    MPIControlAddress   controlAddress;
    MPIControlType      controlType;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motor,
                MOTOR);

        /* Change the ENCODER_FAULT configuration */
    encoderFaultConfigure(motor,
                                                MPIActionABORT,             /* ABORT
on limit  */
                                                EVENT_DURATION);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motor);

    return MPIMessageOK;
}
```

encoder.c -- Continuous display of motor actual position while switching the

```
/* encoder.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/encoder.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Continuous display of motor actual position while switching the
 encoder polarity when a key is pressed.

 Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    encoderMain

argMainRENAME(main, encoder)
#endif

/* Command line arguments and defaults */
long    axisNumber  = -1;
long    motorNumber =  0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

typedef struct _Context *Context;
```

```
typedef struct _Context {
    MPIControl  control;
    MPIAxis     axis;
    MPIMotor    motor;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIMotorConfig  motorConfig;

    long    axisNumber;
    long    motorNumber;

    long    encoderPhase;
} _Context;

long    contextCreate(Context   contextData,
                      long      argc,
                      char      *argv[]);
long    contextDelete(Context   contextData);

void    positionDisplay(double  position);
double  positionGet(Context contextData);

Static _Context ContextObject;

int
    main(int    argc,
         char   *argv[])
{
    Context context;

    long    returnValue;

    context = &ContextObject;

    returnValue =
        contextCreate(context,
                      argc,
                      argv);

    /* Disable amplifier */
    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiMotorAmpEnableSet(context->motor,
                                 FALSE);
    }

    while (returnValue == MPIMessageOK) {
        long    key;

        printf("\nMove the motor and note the direction; it should be %s.\n",
                (context->motorConfig.encoderPhase == 0)
                                ? "normal"
                                : "reversed");

        printf("Press a key to change the encoder phasing, ESC to quit.\n");
```

```
        while ((key = meiPlatformKey(MPIWaitPOLL)) <= 0) {
            positionDisplay(positionGet(context));
        }

        if (key == 0x1b) {
            break;
        }

        context->motorConfig.encoderPhase = (context->motorConfig.encoderPhase == 0);

        returnValue =
            mpiMotorConfigSet(context->motor,
                              &context->motorConfig,
                              NULL);
    }

    returnValue = contextDelete(context);
    meiASSERT(returnValue == MPIMessageOK);

    return ((int)returnValue);
}

long
    contextCreate(Context   context,
                  long      argc,
                  char      *argv[])
{
    long    returnValue;

    long    argIndex;

    memset(context, 0, sizeof(*context));

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &context->controlType,
                   &context->controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber  >= MEIXmpMAX_Axes) ||
```

```
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
    }

    context->motorNumber = motorNumber;

    context->axisNumber =
        (axisNumber < 0)
            ? motorNumber
            : axisNumber;

    context->control =
        mpiControlCreate(context->controlType,
                         &context->controlAddress);
    msgCHECK(mpiControlValidate(context->control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(context->control);
    msgCHECK(returnValue);

    /* Create axis object */
    context->axis =
        mpiAxisCreate(context->control,
                      context->axisNumber);
    msgCHECK(mpiAxisValidate(context->axis));

    /* Create motor object */
    context->motor =
        mpiMotorCreate(context->control,
                       context->motorNumber);
    msgCHECK(mpiMotorValidate(context->motor));

    returnValue =
        mpiMotorConfigGet(context->motor,
                          &context->motorConfig,
                          NULL);
    msgCHECK(returnValue);

    context->encoderPhase = context->motorConfig.encoderPhase;

    return (returnValue);
}

long
    contextDelete(Context   context)
{
    long    returnValue = MPIMessageOK;

    if (context->motor != MPIHandleVOID) {
        /* Set to original configuration */
        context->motorConfig.encoderPhase = context->encoderPhase;
```

```
        returnValue =
            mpiMotorConfigSet(context->motor,
                              &context->motorConfig,
                              NULL);

        if (returnValue == MPIMessageOK) {
            returnValue = mpiMotorDelete(context->motor);

            if (returnValue == MPIMessageOK) {
                context->motor = MPIHandleVOID;
            }
        }
    }

    if (returnValue == MPIMessageOK) {
        if (context->axis != MPIHandleVOID) {
            returnValue = mpiAxisDelete(context->axis);

            if (returnValue == MPIMessageOK) {
                context->axis = MPIHandleVOID;
            }
        }
    }

    if (returnValue == MPIMessageOK) {
        if (context->control != MPIHandleVOID) {
            returnValue = mpiControlDelete(context->control);

            if (returnValue == MPIMessageOK) {
                context->control = MPIHandleVOID;
            }
        }
    }

    return (returnValue);
}

void
    positionDisplay(double  position)
{
    printf("Actual Position: %8.0lf\r",
            position);
}

double
    positionGet(Context context)
{
    long    returnValue;

    double  actualPos;

    returnValue =
        mpiAxisActualPositionGet(context->axis,
                                 &actualPos);
    msgCHECK(returnValue);

    return (actualPos);
```

```
  }
```

event1.c -- Perform a repeated two-axis motion while polling an event manager.

```
/* event1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/event1.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a repeated two-axis motion while polling an event manager.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    event1Main

argMainRENAME(main, event1)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
```

```
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  20000.0,    },
    { 0.0,      0.0,        },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
};

/* motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axisX;      /* X axis */
    MPIAxis     axisY;      /* Y axis */
    MPIMotion   motion;     /* coordinated motion object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    long    returnValue;    /* return value from library */
```

```
    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
```

```
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create X axis object using axis number X_AXIS on controller */
    axisX =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axisX));

    /* Create Y axis object using axis number Y_AXIS on controller */
    axisY =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axisY));

    /* Create motion object using MS number */
    /* Append X axis to motion */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axisX);
    msgCHECK(mpiMotionValidate(motion));

    /* Append Y axis to motion */
    returnValue =
        mpiMotionAxisAppend(motion,
                            axisY);
    msgCHECK(returnValue);

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
```

```
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr =
        mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Flush any existing events */
    returnValue =
        mpiEventMgrFlush(eventMgr);
    msgCHECK(returnValue);

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    printf("Press any key to stop ...\n");

    /* Loop repeatedly */
    index = 0;
    while ((returnValue == MPIMessageOK) &&
           (meiPlatformKey(MPIWaitPOLL) <= 0)) {
        MPIMotionParams motionParams;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        /* Start motion */
```

```
        returnValue =
            mpiMotionStart(motion,
                           motionType,
                           &motionParams);

        switch (returnValue) {
            case MPIMotionMessageERROR: {
                returnValue =
                    mpiMotionAction(motion,
                                    MPIActionRESET);
                msgCHECK(returnValue);

                /* Wait for reset to take effect */
                meiPlatformSleep(2);

                /* FALL THROUGH */
            }
            case MPIMotionMessageNOT_READY: {
                returnValue = MPIMessageOK;
                continue;
            }
            case MPIMotionMessageMOVING: {
                returnValue = MPIMessageOK;
                break;
            }
            case MPIMessageOK:
            default: {
                break;
            }
        }

        /* Collect motion events */
        while (TRUE) {
            MPIEventStatus  eventStatus;

            /* Obtain firmware event(s) (if any) */
            returnValue =
                mpiEventMgrService(eventMgr,
                                   MPIHandleVOID);
            msgCHECK(returnValue);

            /* Poll for motion event */
            returnValue =
                mpiNotifyEventWait(notify,
                                   &eventStatus,
                                   MPIWaitPOLL);

            if (returnValue == MPIMessageOK) {
                if (eventStatus.type == MPIEventTypeMOTION_DONE) {
                    break;
                }
            }
            else {
                meiASSERT(returnValue == MPIMessageTIMEOUT);
```

```
            }
        }

        if (++index >= MOTION_COUNT) {
            index = 0;
        }
    }

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisY);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisX);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

event3.c -- Perform a repeated single-axis motion using command-line-specified axis (default 0).

```
/* event3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/event3.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a repeated single-axis motion using command-line-specified axis (default 0).

Wait for events to be distributed by the event manager in a separate service thread.
 Check status after motion complete, warn of inPosition not set.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    event3Main

argMainRENAME(main, event3)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (1)

/* Command line arguments and defaults */
long            axisNumber      = 0;
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },
```

```
    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0, },
    { 0.0, },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel */
        { 10000.0,      100000.0,   100000.0,   },
    },
    {   /* velocity     accel       decel */
        { 10000.0,      100000.0,   100000.0,   },
    },
};

/* motion parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};


long
    motionDone(MPIMotion    motion,
               MPIStatus    *status);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;        /* motion controller handle */
    MPIAxis     axisList[AXIS_COUNT];   /* axis handle(s) */
    MPIMotion   motion;         /* motion handle */
    MPINotify   notify;         /* event notification object */
    MPIEventMgr eventMgr;       /* event manager handle */

    MPIEventMask    eventMask;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    Service service;

    long    returnValue;    /* return value from library */
```

```
    long    index;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }
```

```
    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object using axisNumber on controller*/
    axisList[0] =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axisList[0]));

    /* Create motion supervisor object using MS number */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create 1-axis motion coordinate system */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axisList);
    msgCHECK(returnValue);

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
```

```
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Loop repeatedly */
    index = 0;
    while ((returnValue == MPIMessageOK) &&
           (meiPlatformKey(MPIWaitPOLL) <= 0)) {
        MPIMotionParams motionParams;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        /* Start motion */
        returnValue =
            mpiMotionStart(motion,
                           motionType,
                           &motionParams);

        fprintf(stderr,
                "mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                motion,
                motionType,
                &motionParams,
                returnValue,
                mpiMessage(returnValue, NULL));

        switch (returnValue) {
            case MPIMotionMessageERROR: {
                returnValue =
                    mpiMotionAction(motion,
                                    MPIActionRESET);
                fprintf(stderr,
                        "mpiMotionAction(0x%x, RESET) returns 0x%x\n",
                        motion,
                        returnValue);
                msgCHECK(returnValue);

                /* FALL THROUGH */
            }
            case MPIMotionMessageNOT_READY: {
```

```
            returnValue = MPIMessageOK;
            continue;
        }
        case MPIMotionMessageMOVING: {
            returnValue = MPIMessageOK;
            break;
        }
        case MPIMessageOK:
        default: {
            break;
        }
    }

    /* Collect motion events */
    while (returnValue == MPIMessageOK) {
        MPIEventStatus  eventStatus;

        /* Wait for motion event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);

        fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);

        if (returnValue == MPIMessageOK) {
            if (eventStatus.type == MPIEventTypeMOTION_DONE) {
                break;
            }
        }
        else {
            break;
        }
    }

    if (returnValue == MPIMessageOK) {
        MPIStatus   status;

        returnValue =
            motionDone(motion,
                       &status);

        if ((returnValue == MPIMessageOK) &&
            (status.state == MPIStateERROR)) {
            if ((status.action == MPIActionABORT) ||
                (status.action == MPIActionE_STOP_ABORT)) {
                double  position[AXIS_COUNT];

                returnValue =
```

```
                        mpiMotionPositionGet(motion,
                                             position,  /* actual */
                                             NULL);     /* command */
                        msgCHECK(returnValue);

                        returnValue =
                            mpiMotionPositionSet(motion,
                                             NULL,      /* actual */
                                             position); /* command */
                        msgCHECK(returnValue);
                    }

                    returnValue =
                        mpiMotionAction(motion,
                                        MPIActionRESET);
                    fprintf(stderr,
                            "mpiMotionAction(0x%x, RESET) returns 0x%x\n",
                            motion,
                            returnValue);
                    msgCHECK(returnValue);
                }
            }

            putchar('\n');

            if (++index >= MOTION_COUNT) {
                index = 0;
            }
        }

        fprintf(stderr,
                "%s exiting: returnValue 0x%x: %s\n",
                argv[0],
                returnValue,
                mpiMessage(returnValue, NULL));

        returnValue = mpiMotionDelete(motion);
        msgCHECK(returnValue);

        for (index = 0; index < AXIS_COUNT; index++) {
            returnValue = mpiAxisDelete(axisList[index]);
            msgCHECK(returnValue);
        }

        returnValue = serviceDelete(service);
        msgCHECK(returnValue);

        returnValue = mpiEventMgrDelete(eventMgr);
        msgCHECK(returnValue);

        returnValue = mpiNotifyDelete(notify);
        msgCHECK(returnValue);

        returnValue = mpiControlDelete(control);
        msgCHECK(returnValue);

        return ((int)returnValue);
    }
```

```
long motionDone(MPIMotion motion,
                MPIStatus *status)
{
    long    returnValue;

    double  actual[AXIS_COUNT];
    double  command[AXIS_COUNT];

    returnValue =
        mpiMotionStatus(motion,
                        status,
                        NULL);
    msgCHECK(returnValue);

    printf("MotionDone: status: state %d action %d eventMask 0x%x\n"
           "\tatTarget %d settled %d %s\n",
            status->state,
            status->action,
            status->eventMask,
            status->atTarget,
            status->settled,
            (status->settled == FALSE)
                    ? "=== NOT SETTLED ==="
                    : "");

    returnValue =
        mpiMotionPositionGet(motion,
                             actual,
                             command);
    msgCHECK(returnValue);

    if (returnValue == MPIMessageOK) {
        long    index;

        /* Display axis positions */
        for (index = 0; index < AXIS_COUNT; index++) {
            printf("\taxis[%d]    position: command %11.3lf\tactual %11.3lf\n",
                    index,
                    command[index],
                    actual[index]);
        }
    }

    return (returnValue);
}
```

frame1.c -- Simple multi-axis frame generated motion profile

```
/* frame1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/frame1.c 4     7/23/01 2:36p Kevinh $";
#endif

/*
:Simple multi-axis frame generated motion profile

The frame is the basic building block to generate trajectory profiles.  Frames
 can be calculated in the host computer or in the controller.  Once downloaded
 into the controller's memory, they are executed sequentially from a FIFO
 buffer.

This sample demonstrates how to generate a simple multi-axis motion profile
 using frames.

The frame structure consists of the following elements:

    MEIXmpMotionType        Mode;       /* MEIXmpMotionTypePATH_OPEN will be used.
    float                   t;          /* Time in samples
    long                    Position;   /* Position in counts
    float                   Velocity;   /* Units are counts per second
    float                   Accel;      /* Units are counts per second^2
    float                   Jerk;
    long                    Control;
    long                    Reserved;

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#define AXIS_COUNT      (2)

#define DISTANCE        (10000)
#define SAMPLE_TIME     (0.005)
#define TIME_SLICE      (0.12)      /* seconds */
#define ACCEL           (2000)
#define POINT_COUNT     (1000)
#define MAX_VEL         (100)

/* Command line arguments and defaults */
long    axisNumber[AXIS_COUNT] = { 0, 1, };
long    motionNumber    = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl          control;        /* motion controller handle */
    MPIAxis             axis[AXIS_COUNT];       /* axis handle(s) */
    MPIMotion           motion;         /* motion handle */
    MEIMotionParams     params;         /* MPI motion parameters */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPINotify       notify;         /* Event notification handle */
    MPIEventMgr     eventMgr;       /* Event manager handle */
    MPIEventMask    eventMask;
    Service         service;        /* Event manager service handle */
    MEIXmpFrame     frame[AXIS_COUNT*POINT_COUNT];


    long            returnValue;    /* return value from library */
    long            point_index;
    double          x, v, a;
    double          x_start[AXIS_COUNT];
    long            index;
    long            motionDone;
    long            argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
```

```
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object(s) */
    for (index = 0; index < AXIS_COUNT; index++){
        axis[index] =
            mpiAxisCreate(control,
                          axisNumber[index]);
        msgCHECK(mpiAxisValidate(axis[index]));
    }

    /* Create motion object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
```

```
                            axis);
    msgCHECK(returnValue);

    /* Get initial command positions */
    for (index = 0; index < AXIS_COUNT; index++) {
        returnValue =
            mpiAxisCommandPositionGet(axis[index],
                                      &x_start[index]);
        msgCHECK(returnValue);
    }

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* Default (max) priority */
                      -1);  /* Default sleep (msec) */
    meiASSERT(service != NULL);

    /* initialize motion parameters */
    params.frame.frame = frame;
    params.frame.point.retain = 1;  /* don't flush frame buffer. */
    params.frame.point.emptyCount = -1; /* start E-Stop if number of frames left
                                           to execute is less than this limit. -1
disables */

    /* first frame */
    point_index = 0;
    for (index = 0; index < AXIS_COUNT; index++){
        frame[point_index].Mode = MEIXmpMotionTypeSTART;
        frame[point_index].t = 0.0;
        frame[point_index].Position = 0;
```

```
        frame[point_index].Velocity = 0.0;
        frame[point_index].Accel = 0.0;
        frame[point_index].Jerk = 0.0;
        frame[point_index].Control = 0;
        frame[point_index].Reserved = 0;
        point_index++;
    }

    /* initial trajectory values */
    x = 0.0;
    v = 0.0;
    a = ACCEL;

    /* calculate the motion profile frames */
    while (x < DISTANCE) {
        for (index = 0; index < AXIS_COUNT; index++){
            frame[point_index].Mode = MEIXmpMotionTypeS_CURVE;
            frame[point_index].t = TIME_SLICE / SAMPLE_TIME; /* samples */
            frame[point_index].Position = (long)(x + x_start[index]);
            frame[point_index].Velocity = (float)(v * SAMPLE_TIME);
            frame[point_index].Accel = (float)(a * SAMPLE_TIME * SAMPLE_TIME);
            frame[point_index].Jerk = 0.0;
            frame[point_index].Control = 0;
            frame[point_index].Reserved = 0;
            point_index++;
        }

        if (v < MAX_VEL) {
            a = ACCEL;
        }
        else {
            a = 0.0;
        }
        x += v * TIME_SLICE + 0.5 * a * TIME_SLICE * TIME_SLICE;
        v += a * TIME_SLICE;
    }

    /* calculate last frame to set final target conditions exactly */
    for (index = 0; index < AXIS_COUNT; index++){
        frame[point_index].Mode = MEIXmpMotionTypeS_CURVE;
        frame[point_index].t = 2; /* 2 time samples are usually put in this last
 frame */
        frame[point_index].Position = (long)(x + x_start[index]);
        frame[point_index].Velocity = 0.0;
        frame[point_index].Accel = 0.0;
        frame[point_index].Jerk = 0.0;
        frame[point_index].Control = 0;
        frame[point_index].Reserved = 0;
        point_index++;
    }

    /* null frame (placeholder) */
    for (index = 0; index < AXIS_COUNT; index++){
        frame[point_index].Mode = 0;
        frame[point_index].t = 0.0;
        frame[point_index].Position = (long)(x + x_start[index]);
```

```
        frame[point_index].Velocity = 0.0;
        frame[point_index].Accel = 0.0;
        frame[point_index].Jerk = 0.0;
        frame[point_index].Control = 0;
        frame[point_index].Reserved = 0;
        point_index++;
    }

    params.frame.point.final = TRUE;
    params.frame.pointCount = point_index; /* point_index is the total number of
frames for all axes */

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       (MPIMotionType)(MEIMotionTypeFRAME),
                       (MPIMotionParams*)(&params));
    fprintf(stderr,
            "mpiMotionStart returns 0x%x: %s\n",
            returnValue,
            mpiMessage(returnValue, NULL));
    msgCHECK(returnValue);


    /* Collect motion events */
    motionDone = FALSE;
    while (motionDone != TRUE) {
        MPIEventStatus  eventStatus;

        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);
        msgCHECK(returnValue);

        switch(eventStatus.type) {
            case MPIEventTypeMOTION_DONE: {
                motionDone = TRUE;
                break;
            }
            default: {
                break;
            }
        }

        fprintf(stderr,
                "mpiNotifyEventWait() returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        msgCHECK(returnValue);
    }

    /* Delete Objects */
```

```
    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (index = 0; index < AXIS_COUNT; index++){
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

home1.c -- Simple Homing routine that captures the hardware position, sets the origin

```
/* home1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/home1.c 9     7/18/01 9:32a Kevinh $"
#endif

#if defined(ARG_MAIN_RENAME)
#define main    home1Main

argMainRENAME(main, home1)
#endif

/*
:Simple Homing routine that captures the hardware position, sets the origin
  and moves back to home.

home1.c allows the user to trigger his home off an input or index
 pulse, capture the hardware position, set the origin and then move back
 to that home position.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define CapturesPerMotor    (2) /* Default Capture configuration */

#define ACTIVE_FALLING_EDGE     (0)
#define ACTIVE_RAISING_EDGE     (1)

/* MPI Object numbers */
#define MOTION_NUMBER   (0)
```

```
#define AXIS_NUMBER     (0)
#define MOTOR_NUMBER    (0)

/* Motion Parameters */
#define VELOCITY        (5000)  /* Move velocity     */
#define ACCELERATION    (10000) /* Move acceleration */
#define DECELERATION    (10000) /* Move deceleration */

/* Capture Parameters */
#define CAPTURE_EDGE        (ACTIVE_FALLING_EDGE) /* Capture on falling edge */

/* CAPTURE_TRIGGER can be MEIMotorInputHOME or MEIMotorInputINDEX */
#define CAPTURE_TRIGGER     (MEIMotorInputHOME)   /* Capture on home pulse */


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr, "usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber,
                 MPIMotor           *motor,
                 long                motorNumber,
                 MPICapture         *capture,
                 long                captureNumber)
{
    long           returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
```

```
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        *axis);
    msgCHECK(mpiMotionValidate(*motion));

    /* Create motor object */
    *motor =
        mpiMotorCreate(*control,
                        motorNumber);
    msgCHECK(mpiMotorValidate(*motor));

    /* Create capture object */
    *capture =
        mpiCaptureCreate(*control,
                         captureNumber);
    msgCHECK(mpiCaptureValidate(*capture));
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis,
                    MPIMotor        *motor,
                    MPICapture      *capture)
{
    long    returnValue;


    /* Delete capture object */
    returnValue =
        mpiCaptureDelete(*capture);
    msgCHECK(returnValue);

    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);
```

```
    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    /* Delete axis object */
    returnValue =
        mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Calculate default capture number for axisNumber */
long captureNumber(long     motorNumber)
{
    return  ((motorNumber/MEIXmpMotorsPerBlock) * MEIXmpMaxLatches) +
            ((motorNumber % MEIXmpMotorsPerBlock) * CapturesPerMotor);
}


/*
   Configure caputure object

   edge: 0 for falling edge, 1 for rising edge
*/
void configureCapture(MPICapture    capture,
                      MEIMotorInput trigger,
                      long          edge)
{
    MPICaptureConfig    captureConfig;
    long                returnValue;


    /* Disable capture */
    returnValue =
        mpiCaptureArm(capture,
                      FALSE);
    msgCHECK(returnValue);

    /* Read caputre configuration */
    returnValue =
        mpiCaptureConfigGet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);

    /* Set capture parameters */
    captureConfig.trigger.mask    = trigger;
    captureConfig.trigger.pattern = edge ? trigger : 0;
```

```
    /* Write caputre configuration */
    returnValue =
        mpiCaptureConfigSet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);
}


/* Configure Home Event action */
void configureHomeAction(MPIMotor   motor,
                         MPIAction  action,
                         long       activeHigh)
{
    MPIMotorEventConfig eventConfig;
    MPIEventMask        eventMask;
    long                returnValue;


    /* Read home event configuration */
    returnValue =
        mpiMotorEventConfigGet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Configure Home Event action */
    eventConfig.action = action;
    eventConfig.trigger.polarity = activeHigh;

    /* Write home event configuration */
    returnValue =
        mpiMotorEventConfigSet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Reset Home Event */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskSET(eventMask, MPIEventTypeHOME);
    returnValue =
        mpiMotorEventReset(motor,
                           eventMask);
    msgCHECK(returnValue);
}


/* Command simple trapezoidal motion */
void simpleVelocityMove(MPIMotion   motion,
                        double      velocity,
                        double      acceleration,
```

```
                        double      deceleration)
{
    MPIMotionParams params;      /* Motion parameters      */
    MPITrajectory   trajectory;  /* Trajectory information */

    long            returnValue; /* MPI library return value */


    /* Setup trajectory structure */
    trajectory.velocity     = velocity;
    trajectory.acceleration = acceleration;
    trajectory.deceleration = deceleration;
    trajectory.jerkPercent  = 0.0;          /* Trapezoidal profile */

    /* Setup parameters structure */
    params.velocity.trajectory  = &trajectory;

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeVELOCITY,
                       &params);
    msgCHECK(returnValue);
}


/* Command simple trapezoidal motion */
void simpleTrapMove(MPIMotion   motion,
                    double      goalPosition,
                    double      velocity,
                    double      acceleration,
                    double      deceleration)
{
    MPIMotionParams params;      /* Motion parameters      */
    MPITrajectory   trajectory;  /* Trajectory information */

    long            returnValue; /* MPI library return value */


    /* Setup trajectory structure */
    trajectory.velocity     = velocity;
    trajectory.acceleration = acceleration;
    trajectory.deceleration = deceleration;

    /* Setup parameters structure */
    params.trapezoidal.trajectory   = &trajectory;
    params.trapezoidal.position     = &goalPosition;

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
                       &params);
    msgCHECK(returnValue);
```

```
}


/* Display caputure status while waiting for the motion to be done */
void displayCaptureStatusUntilMotionDone(MPIMotion          motion,
                                         MPICapture         capture,
                                         MPICaptureStatus   *captureStatus)
{
    MPIStatus   status;
    long        motionDone;
    long        returnValue;


    /* Poll status until motion done */
    motionDone = FALSE;
    while (motionDone == FALSE) {

        /* Get the capture status */
        returnValue =
            mpiCaptureStatus(capture,
                             captureStatus,
                             NULL);
        msgCHECK(returnValue);

        printf("CaptureState:0x%x\r",
                captureStatus->state);

        /* Get the motion supervisor status */
        returnValue =
            mpiMotionStatus(motion,
                            &status,
                            NULL);
        msgCHECK(returnValue);

        switch (status.state) {
            case MPIStateSTOPPING:
            case MPIStateMOVING: {
                /* Sleep for 20ms and give up control to other threads */
                meiPlatformSleep(20);
                break;
            }
            case MPIStateIDLE:
            case MPIStateERROR:
            case MPIStateSTOPPING_ERROR: {
                /* Motion is done */
                motionDone = TRUE;
                break;
            }
            default: {
                /* Unknown State */
                fprintf(stderr, "Unknown state from mpiMotionStatus.\n");
                msgCHECK(MPIMessageFATAL_ERROR);
                break;
            }
```

```
        }
    }
    /* Display latched position */
    printf("\nLatched Home Position = %.0lf\n",
           captureStatus->latch[0]);
}


int main(int     argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;
    MPIAxis             axis;
    MPIMotor            motor;
    MPICapture          capture;
    MPICaptureStatus    captureStatus;

    long    returnValue;        /* Return value from library */


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                &axis,
                AXIS_NUMBER,
                &motor,
                MOTOR_NUMBER,
                &capture,
                captureNumber(MOTOR_NUMBER));

    /* Configure capture */
    configureCapture(capture,
                     CAPTURE_TRIGGER,
                     CAPTURE_EDGE);

    /* Configure Home Event action to stop */
    configureHomeAction(motor,
                        MPIActionSTOP,
                        TRUE);          /* Active High */

    /* Arm the capture */
    returnValue =
```

```
        mpiCaptureArm(capture,
                      TRUE);
    msgCHECK(returnValue);

    printf("Looking for home...\n");

    /* Start a velocity move */
    simpleVelocityMove(motion,
                       VELOCITY,
                       ACCELERATION,
                       DECELERATION);

    /* Display caputure status while waiting for the motion to be done */
    displayCaptureStatusUntilMotionDone(motion,
                                        capture,
                                        &captureStatus);

    /* Set origin to home position */
    returnValue =
        mpiAxisOriginSet(axis,
                         captureStatus.latch[0]);
    msgCHECK(returnValue);

    printf("Moving back to origin.\n");

    /* Configure Home Event action to none */
    configureHomeAction(motor,
                        MPIActionNONE,
                        TRUE);

    /* Move back to home */
    simpleTrapMove(motion,
                   0.0,     /* Go back to home position */
                   VELOCITY,
                   ACCELERATION,
                   DECELERATION);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis,
                   &motor,
                   &capture);

    return ((int)returnValue);
}
```

initFlsh.c -- Controller initialization and automatic firmware download

```
/* initFlsh.c */

/* Copyright(c) 1991-2002 Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIUtilRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/initFlsh.c 4     7/23/01 2:36p Kevinh $";
#endif

/*

:Controller initialization and automatic firmware download

  ********************************* Warning ***********************************
  This application will overwrite the controller configuration that is saved
  to the Flash memory.  If connected to hardware this could cause motors to
  be enabled or to run away.  Do NOT use without proper knowledge of the system
  *****************************************************************************

This sample code demonstrates how to download Firmware to the XMP-Series
 controller.  The Firmware can also be downloaded using one of the utilities
 provided by MEI.  This sample application requires the user to specify the
 location and filename for a valid, matching Firmware file and optional
 FPGA file(s).  During initialization the controller's firmware version,
 revision, option and user version numbers are checked.  If the controller's
 version doesn't match the expected version, the firmware is downloaded to
 the controller's flash memory.

Note: The flash utility program uses the same code to download firmware to the
 flash memory.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

/* Set desired Firmware version. these values could be read from a file */
```

```
#define FW_VERSION          332
#define FW_REVISION         'A'
#define FW_SUBREVISION      2
#define FW_OPTION           0
#define FW_VERSION_USER     0   /* user configurable */

#if defined(ARG_MAIN_RENAME)
#define main    initFlshMain

argMainRENAME(main, initFlsh)
#endif

#define FIRMWARE    "c:\\xmp332a2.bin"  /* this string could be read from a file */
#define FPGA0       (NULL)              /* specify NULL for default FPGA image */
#define FPGA1       (NULL)
#define FPGA2       (NULL)

long initXmp(MPIControl control, MEIFlashFiles *fileNames);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl          control;

    long    returnValue;
    long    argIndex;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MEIFlashFiles       flashFiles;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* copy file names into structure */
    if (FIRMWARE != NULL) {
        strcpy(flashFiles.binFile, FIRMWARE);
    }
    else {
        flashFiles.binFile[0] = '\0';
    }
    if (FPGA0 != NULL) {
        strcpy(flashFiles.FPGAFile[0], FPGA0);
    }
    else {
```

```
        flashFiles.FPGAFile[0][0] = '\0';
    }
    if (FPGA1 != NULL) {
        strcpy(flashFiles.FPGAFile[1], FPGA1);
    }
    else {
        flashFiles.FPGAFile[1][0] = '\0';
    }
    if (FPGA2 != NULL) {
        strcpy(flashFiles.FPGAFile[2], FPGA2);
    }
    else {
        flashFiles.FPGAFile[2][0] = '\0';
    }

    /* Initialize the controller, load flash if firmware is different */
    returnValue =
        initXmp(control,
                &flashFiles);   /* firmware and FPGA file name(s) */
    msgCHECK(returnValue);

    /* Delete object handles */
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long versionCheck(MPIControl control)
{
    MPIControlConfig    controlConfig;
    MEIControlVersion   version;

    long returnValue;

    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);

    returnValue =
        meiControlVersionGet(control,
                             &version);

    if ((version.xmp.firmware.version != FW_VERSION)
        || (version.xmp.firmware.revision != FW_REVISION)
        || (version.xmp.firmware.subRevision != FW_SUBREVISION)
        || (version.xmp.firmware.option != FW_OPTION)
        || (controlConfig.userVersion != FW_VERSION_USER)) {

        meiPlatformConsole("ERROR: Controller firmware does not match expected
version.\n"
                           " Version:  %d%c%d option:%d user:%d\n"
                           " Expected: %d%c%d option:%d user:%d\n\n",
                           version.xmp.firmware.version,
                           version.xmp.firmware.revision,
                           version.xmp.firmware.subRevision,
```

```
                            version.xmp.firmware.option,
                            controlConfig.userVersion,
                            FW_VERSION,
                            FW_REVISION,
                            FW_SUBREVISION,
                            FW_OPTION,
                            FW_VERSION_USER);

        returnValue = MEIControlMessageFIRMWARE_VERSION;
    }

    /* display check result */
    meiPlatformConsole("Firmware Version Check: %s\n", mpiMessage(returnValue,
NULL));

    return (returnValue);
}

long firmwareLoad(MPIControl control, MEIFlashFiles *fileNames)
{
    MEIFlash            flash;
    MEIFlashFiles       loadedFiles;
    MEIControlVersion   version;

    long                MB0 = -1;
    long                MB1 = -1;
    long                update20Khz = 0;

    long        returnValue;

    returnValue =
        mpiControlValidate(control);

    if (returnValue == MPIMessageOK) {
        /* fake a valid version so we can still use the MPI routines */
        returnValue =
            meiControlVersionGet(control,
                                 &version);

        if (returnValue == MPIMessageOK) {
            memset(&version.xmp.firmware,
                   0,
                   sizeof(version.xmp.firmware));

            version.xmp.firmware.version = version.mpi.firmware.version;
            version.xmp.firmware.option  = version.mpi.firmware.option;

            returnValue =
                meiControlVersionSet(control,
                                     &version);
        }
    }

    if (returnValue == MPIMessageOK) {
        /* Create Flash object */
        flash =
            meiFlashCreate(control);
```

```
        returnValue = meiFlashValidate(flash);
    }

    if (returnValue == MPIMessageOK) {
        meiPlatformConsole("Loading flash memory from \"%s\" ...\n",
                                fileNames->binFile);
    }

    if (returnValue == MPIMessageOK) {
        /* Load Firmware and FPGAs from file(s) */
        returnValue =
            meiFlashMemoryFromFile(flash,
                                fileNames,
                                &loadedFiles);
    }

    if (returnValue == MPIMessageOK) {
        meiPlatformConsole("Code loaded and verified from \"%s\".\n",
                           loadedFiles.binFile);

        meiPlatformConsole("FPGAs loaded and verified from\n"
                           "%s\n"
                           "%s\n"
                           "%s\n\n",
                           loadedFiles.FPGAFile[0],
                           loadedFiles.FPGAFile[1],
                           loadedFiles.FPGAFile[2]);

    }
    else {
        meiPlatformConsole("Can't load flash from \"%s\"\n"
                           "\"%s\"\n"
                           "\"%s\"\n"
                           "\"%s\"\n"
                           "(%s)\n",
                           loadedFiles.binFile,
                           loadedFiles.FPGAFile[0],
                           loadedFiles.FPGAFile[1],
                           loadedFiles.FPGAFile[2],
                           mpiMessage(returnValue, NULL));
    }

    if (returnValue == MPIMessageOK) {
        returnValue = mpiControlReset(control);
        msgCHECK(returnValue);
    }

    if (returnValue == MPIMessageOK) {
        MEIControlSocketInfo socketInfo;
        MEIControlRipTideConfig config;

        returnValue = meiControlSocketInfoGet(control,
                                              &socketInfo);
        msgCHECK(returnValue);

        /* Get default RipTide configuration */
        returnValue = meiControlRipTideDefaultGet(control,
```

```
                                                &socketInfo,
                                                &config);
    msgCHECK(returnValue);

    /* Configure RipTide */
    if (MB0 != -1) {
        config.motionBlocks[0] = MB0;
    }

    if (MB1 != -1) {
        config.motionBlocks[1] = MB1;
    }

    config.update20khz = update20Khz;

    if (config.update20khz) {
        MPIControlConfig controlConfig;

        config.motionBlocks[0] = 1;
        config.motionBlocks[1] = 0;

        /* Set Sample Rate */
        returnValue = mpiControlFlashConfigGet(control,
                                                NULL,
                                                &controlConfig,
                                                NULL);
        msgCHECK(returnValue);

        if (returnValue == MPIMessageOK) {
            controlConfig.sampleRate = 20000;

            returnValue = mpiControlFlashConfigSet(control,
                                                   NULL,
                                                   &controlConfig,
                                                   NULL);
            msgCHECK(returnValue);
        }
    }

    if (returnValue == MPIMessageOK) {
        meiPlatformConsole("Configuring RipTide\n");

        returnValue = meiControlFlashRipTideConfigSet(control,
                                                      flash,
                                                      &config);

        if (config.update20khz) {
            meiPlatformConsole("Reconfigured RipTide:\n"
                               "Configured for 20Khz operation\n"
                               "Main Board has 1 Motion Block\n"
                               "Expansion Board has 0 Motion Blocks\n\n");
        }
        else {
            meiPlatformConsole("Reconfigured RipTide:\n"
                               "Main Board has %d Motion Blocks\n"
                               "Expansion Board has %d Motion Blocks\n\n",
                               config.motionBlocks[0],
```

```
                                    config.motionBlocks[1]);
            }
        }
    }

    /* clean up */
    if (returnValue == MPIMessageOK) {
        returnValue = meiFlashDelete(flash);
    }

    /* re-initialize controller */
    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiControlInit(control);
    }

    /* check version values */
    if (returnValue == MPIMessageOK) {
        returnValue =
            versionCheck(control);
    }

    return returnValue;
}

long initXmp(MPIControl control, MEIFlashFiles *fileNames)
{
    long returnValue;

    /* Initialize the controller */
    returnValue =
        mpiControlInit(control);

    /* check init returnValue */
    if ((returnValue == MEIControlMessageFIRMWARE_INVALID) ||
        (returnValue == MEIControlMessageFIRMWARE_VERSION) ||
        (returnValue == MEIControlMessageFIRMWARE_VERSION_NONE)) {
        /* display warning */
        meiPlatformConsole("WARNING: %s\n\n",
                   mpiMessage(returnValue, NULL));

        /* load firmware */
        returnValue =
            firmwareLoad(control, fileNames);
    }
    else {
        if (returnValue == MPIMessageOK) {
            /* check firmware version, revision, option, and user version */
            returnValue =
                versionCheck(control);

            if (returnValue == MEIControlMessageFIRMWARE_VERSION) {
                /* load firmware */
                returnValue =
                    firmwareLoad(control, fileNames);
            }
        }
```

```
    }

    return (returnValue);
}
```

limitSw1.c -- Configure positive and negative hardware limit inputs.

```
/* limitsw1.c */

/* Copyright(c) 1991-2001 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/limitsw1.c 5     7/18/01 9:32a Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    limitsw1Main
argMainRENAME(main, limitsw1)
#endif

/*

:Configure positive and negative hardware limit inputs.

A motor's positive and negative hardware limit inputs have four configurations:

  1) Event Action   (any MPIAction such as MPIActionE_STOP)
  2) Event Trigger  (a trigger polarity, active HIGH or LOW)
  3) Direction Flag (TRUE will cause the command direction of motion to
                         qualify the events, FALSE will ignore direction,
                         based solely on the limit input state)
  4) Duration       (requires the limit condition to exist for
                         a programmable number of seconds before an
                         event will occur)

Events are configured using MPIMotorEventConfigGet/Set(...).

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#define MOTOR           (0)
#define EVENT_DURATION  (0.010)     /* seconds */


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


void limitConfigure(MPIMotor        motor,
                    MPIEventType    eventType,
                    MPIAction       action,
                    long            polarity,
                    long            direction,
                    double          duration)
{
    MPIMotorEventConfig eventConfig;
    long                returnValue;


    /* Get the current limit configuration */
    returnValue =
        mpiMotorEventConfigGet(motor,
                               eventType,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Set trigger level to Active Low (FALSE) or Active High (TRUE) */
    eventConfig.trigger.polarity = polarity;

    /* Use commanded velocity direction as a trigger qualifier. (TRUE/FALSE) */
    eventConfig.direction = direction;

    /* Wait duration seconds before triggering */
    eventConfig.duration = (float) duration;

    /* Configure motor to perform action upon limit */
    eventConfig.action = action;
```

```
    /* Set the new limit configuration */
    returnValue =
        mpiMotorEventConfigSet(motor,
                               eventType,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotor           *motor,
                 long                motorNumber)
{
    long returnValue;


    /* Obtain a Control handle */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize the controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Obtain a Motor handle */
    *motor =
        mpiMotorCreate(*control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(*motor));
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIMotor    *motor)
{
    long returnValue;

    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);

    /* Delete control object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
```

```
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;
    MPIMotor            motor;
    MPIControlAddress   controlAddress;
    MPIControlType      controlType;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motor,
                MOTOR);

    /* Change the LIMIT_HW_POS configuration */
    limitConfigure(motor,
                   MPIEventTypeLIMIT_HW_POS,    /* +HW limit        */
                   MPIActionE_STOP,             /* E-STOP on limit  */
                   TRUE,                        /* Active High      */
                   TRUE,     /* Use velocity as a trigger qualifier */
                   EVENT_DURATION);

    /* Change the LIMIT_HW_POS configuration */
    limitConfigure(motor,
                   MPIEventTypeLIMIT_HW_NEG,    /* -HW limit        */
                   MPIActionE_STOP,             /* E-STOP on limit  */
                   TRUE,                        /* Active High      */
                   TRUE,     /* Use velocity as a trigger qualifier */
                   EVENT_DURATION);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motor);

    return MPIMessageOK;
}
```

mboard1.c -- Setup multiple controllers, clear positions and perform a two-point motion.

```
/* mboard1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software contains proprietary and confidential information of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent
 */
#if defined(MEI_RCS)
    static const char MEIAppRCS[] =
        "$Header: /MainTree/XMPLib/XMP/app/mboard1.c 10    7/23/01 2:36p Kevinh $";
#endif
/*

:Setup multiple controllers, clear positions and perform a two-point motion.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    mboard1Main

argMainRENAME(main, mboard1)
#endif

#define BOARD_COUNT (2)

#define MOTION_COUNT    (1)
#define AXIS_COUNT      (1)

/* Command line arguments and defaults */
long            controlNumber   = 0;
long            axisNumber      = 0;
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-control", ArgTypeLONG,    &controlNumber, },
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },
```

```
    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0, },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel */
        { 10000.0,      100000.0,   100000.0,   },
    },
};

/* motion parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl          controller[BOARD_COUNT];    /* Array of control handles   */
    MPIMotion           motion[BOARD_COUNT];        /* Array of motion  handles   */
    MPIAxis             axis[BOARD_COUNT];          /* Array of axis    handles   */

    MPIControlAddress   controlAddress;

    long    returnValue;
    long    argIndex = 0;
    long    index;

    /* Parse command line for application-specific arguments */
    while (argIndex <= argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }
```

```
    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc - 1) ||    /* minus application name */
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /*
        Initialize each controller with default settings,
        specifying the board's controller number.
     */
    for (index = 0; index < BOARD_COUNT; index++) {
        controlAddress.number = controlNumber + index;  /* setting controller number
*/

        controller[index] =
            mpiControlCreate(MPIControlTypeDEFAULT,
                             &controlAddress);
        msgCHECK(mpiControlValidate(controller[index]));
    }

    /* Initialize the controllers */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiControlInit(controller[index]);

        if (returnValue != MPIMessageOK) {
            fprintf(stderr,
                    "Controller #%d : mpiControlInit(0x%x) returns 0x%x: %s\n",
                    index,
                    controller[index],
                    returnValue,
```

```
                mpiMessage(returnValue, NULL));

            exit(1);
        }
    }

    /*  Obtain Axis and Motion handle for each board */
    for (index = 0; index < BOARD_COUNT; index++) {
        axis[index] =
            mpiAxisCreate(controller[index],
                          axisNumber);
        msgCHECK(mpiAxisValidate(axis[index]));

        /* Create motion object, append axis */
        motion[index] =
            mpiMotionCreate(controller[index],
                            motionNumber,
                            axis[index]);
        msgCHECK(mpiMotionValidate(motion[index]));
    }

    /* Clear the command and actual positions on each controller */
    for (index = 0; index < BOARD_COUNT; index++) {
        double command;

        returnValue =
            mpiAxisCommandPositionGet(axis[index],
                                      &command);
        msgCHECK(returnValue);

        returnValue =
            mpiAxisOriginSet(axis[index],
                             command);
        msgCHECK(returnValue);

        /* clear position error */
        returnValue =
            mpiAxisActualPositionSet(axis[index],
                                     0.0);
        msgCHECK(returnValue);
    }

    /* Sequentially Start motion */
    for (index = 0; index < BOARD_COUNT; index++) {
        MPIMotionParams motionParams;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[0];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[0];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[0];
```

```
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        /* Start motion */
        returnValue =
            mpiMotionStart(motion[index],
                           motionType,
                           &motionParams);

        fprintf(stderr,
                "mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                motion[index],
                motionType,
                &motionParams,
                returnValue,
                mpiMessage(returnValue, NULL));

        switch (returnValue) {
            case MPIMotionMessageERROR: {
                returnValue =
                    mpiMotionAction(motion[index],
                                    MPIActionRESET);
                fprintf(stderr,
                        "mpiMotionAction(0x%x, RESET) returns 0x%x\n",
                        motion[index],
                        returnValue);
                msgCHECK(returnValue);

                /* FALL THROUGH */
            }
            case MPIMotionMessageNOT_READY: {
                returnValue = MPIMessageOK;
                continue;
            }
            case MPIMotionMessageMOVING: {
                returnValue = MPIMessageOK;
                break;
            }
            case MPIMessageOK:
            default: {
                break;
            }
        }
    }

    /* Delete the Motion handles */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiMotionDelete(motion[index]);
        msgCHECK(returnValue);
    }

    /* Delete the Axis handles */
```

```
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    /* Delete the CONTROL handles */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiControlDelete(controller[index]);
        msgCHECK(returnValue);
    }

    return ((int)returnValue);
}
```

mboard2.c -- Two point motion on multiple controllers with an event manager in polling mode.

```
/* mboard2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software contains proprietary and confidential information of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent
 */
#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/mboard2.c 12    7/23/01 2:36p Kevinh $";
#endif
/*

:Two point motion on multiple controllers with an event manager in polling mode.

Creates a two point motion sequentially on multiple controller boards.
 One event manager is used for all controllers.  The event manager is polled
 to retrieve MotionDone events from each motion.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    mboard2Main

argMainRENAME(main, mboard2)
#endif

#define BOARD_COUNT (2)

#define MOTION_COUNT    (1)
#define AXIS_COUNT      (1)

/* Command line arguments and defaults */
long            controlNumber   = 0;
long            axisNumber      = 0;
long            motionNumber    = 0;
```

```
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-control", ArgTypeLONG,    &controlNumber, },
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0 },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel */
        { 10000.0,      100000.0,   100000.0,   },
    },
};

/* motion parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl      controller[BOARD_COUNT];/* Array of control handles */
    MPIMotion       motion[BOARD_COUNT];    /* Array of motion handles */
    MPIAxis         axis[BOARD_COUNT];      /* Array of axis handles */
    MPINotify       notify[BOARD_COUNT];    /* Array of notify handles */
    MPIEventMgr     eventMgr;   /* Array of event manager handles */

    MPIEventMask    eventMask;

    MPIControlAddress   address;

    long    returnValue;
    long    argIndex=0;
    long    index;

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;
```

```
        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc - 1) ||    /* minus application name */
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /*
        Initialize each controller with default settings,
        specifying the board's controller number.
     */
    for (index = 0; index < BOARD_COUNT; index++) {
        address.number = controlNumber + index; /* set controller number */

        controller[index] =
            mpiControlCreate(MPIControlTypeDEFAULT,
                             &address);
```

```
        msgCHECK(mpiControlValidate(controller[index]));
    }

    /* Initialize the controllers */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiControlInit(controller[index]);

        if (returnValue != MPIMessageOK) {
            fprintf(stderr,
                    "Controller #%d : mpiControlInit(0x%x) returns 0x%x: %s\n",
                    index,
                    controller[index],
                    returnValue,
                    mpiMessage(returnValue, NULL));

            exit(1);
        }
    }

    /* Obtain Axis and Motion handle for each board */
    for (index = 0; index < BOARD_COUNT; index++) {
        axis[index] =
            mpiAxisCreate(controller[index],
                          axisNumber);
        msgCHECK(mpiAxisValidate(axis[index]));

        /* Create motion object, append axis */
        motion[index] =
            mpiMotionCreate(controller[index],
                            motionNumber,
                            axis[index]);
        msgCHECK(mpiMotionValidate(motion[index]));
    }

    /* Setup one Event Manager for all boards */

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(controller[0]);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    for (index = 1; index < BOARD_COUNT; index++) {
        /* Append additional controllers to the event manager */
        returnValue =
            mpiEventMgrControlAppend(eventMgr,
                                     controller[index]);
        msgCHECK(returnValue);
    }

    /* Flush any existing events */
    returnValue = mpiEventMgrFlush(eventMgr);
    msgCHECK(returnValue);

    /* Setup Notify Object */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskSET(eventMask, MPIEventTypeMOTION_DONE);
```

```
    for (index = 0; index < BOARD_COUNT; index++) {
        /* Request notification of Motion Done events from motion */
        returnValue =
            mpiMotionEventNotifySet(motion[index],
                                    eventMask,
                                    NULL);
        msgCHECK(returnValue);

        /* Create event notification object for motion */
        notify[index] =
            mpiNotifyCreate(eventMask,
                            motion[index]);
        msgCHECK(mpiNotifyValidate(notify[index]));

        /* Add notify to event manager's list */
        returnValue =
            mpiEventMgrNotifyAppend(eventMgr,
                                    notify[index]);
        msgCHECK(returnValue);
    }

    /* Clear the command and actual positions on each controller */
    for (index = 0; index < BOARD_COUNT; index++) {
        double command;

        returnValue =
            mpiAxisCommandPositionGet(axis[index],
                                      &command);
        msgCHECK(returnValue);

        returnValue =
            mpiAxisOriginSet(axis[index],
                             command);
        msgCHECK(returnValue);

        /* clear position error */
        returnValue =
            mpiAxisActualPositionSet(axis[index],
                                     0.0);
        msgCHECK(returnValue);
    }

    /* Sequentially start motion */
    for (index = 0; index < BOARD_COUNT; index++) {
        MPIMotionParams motionParams;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[0];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[0];
                break;
```

```
        }
        case MPIMotionTypeVELOCITY: {
            motionParams.velocity = velocity[0];
            break;
        }
        default: {
            meiASSERT(FALSE);
            break;
        }
    }

    /* Start motion */
    returnValue =
        mpiMotionStart(motion[index],
                       motionType,
                       &motionParams);

    fprintf(stderr,
            "mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
            motion[index],
            motionType,
            &motionParams,
            returnValue,
            mpiMessage(returnValue, NULL));

    switch (returnValue) {
        case MPIMotionMessageERROR: {
            returnValue =
                mpiMotionAction(motion[index],
                                MPIActionRESET);
            fprintf(stderr,
                    "mpiMotionAction(0x%x, RESET) returns 0x%x\n",
                    motion[index],
                    returnValue);
            msgCHECK(returnValue);
            /* FALL THROUGH */
        }
        case MPIMotionMessageNOT_READY: {
            returnValue = MPIMessageOK;
            continue;
        }
        case MPIMotionMessageMOVING: {
            returnValue = MPIMessageOK;
            break;
        }
        case MPIMessageOK:
        default: {
            break;
        }
    }


    /* Collect motion events (Polling Event Manager)*/
    while (TRUE) {
        MPIEventStatus  eventStatus;
```

```
            /* Obtain firmware event(s) (if any) */
            returnValue =
                mpiEventMgrService(eventMgr,
                                   MPIHandleVOID);
            msgCHECK(returnValue);

            /* Poll for motion event */
            returnValue =
                mpiNotifyEventWait(notify[index],
                                   &eventStatus,
                                   MPIWaitPOLL);

            if (returnValue == MPIMessageTIMEOUT) {
                /* no events have happend yet */
                returnValue = MPIMessageOK;
            }

            if (returnValue == MPIMessageOK) {
                if (eventStatus.type == MPIEventTypeMOTION_DONE) {
                    /* Caught the motion done event */
                    printf("Motion Done: type %d source 0x%x info 0x%x\n",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
            }
            else {
                printf("%d\n",returnValue);
                meiASSERT(FALSE);
            }
        }
    }

    /* Delete the EventMgr handle (must happen before removing Notify) */
    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    /* Delete the Notify handles */
    for (index = 0; index < BOARD_COUNT ;index++) {
        returnValue = mpiNotifyDelete(notify[index]);
        msgCHECK(returnValue);
    }

    /* Delete the Motion handles */
    for ( index = 0; index < BOARD_COUNT ;index++) {
        returnValue = mpiMotionDelete(motion[index]);
        msgCHECK(returnValue);
    }

    /* Delete the Axis handles */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
```

```
    }

    /* Delete the CONTROL handles */
    for (index = 0; index < BOARD_COUNT; index++) {
        returnValue = mpiControlDelete(controller[index]);
        msgCHECK(returnValue);
    }

    return ((int)returnValue);
}
```

motGate1.c -- Load and trigger a point to point motion using a control Gate.

```
/* motGate1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/motgate1.c 5     7/18/01 9:32a Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    motGate1Main
argMainRENAME(main, motGate1)
#endif

/*

:Load and trigger a point to point motion using a control Gate.

This sample program demonstrates how to use the motion HOLD attribute and
 a control Gate, to pre-load a single axis trapezoidal profile motion.

The XMP-Series controller supports a motion HOLD attribute, which is useful
 for preloading and triggering motion profiles.  One or more motion supervisors
 can be started from the same HOLD conditions.  When multiple motion supervisors
 are triggered by the same HOLD conditions, the individual motion profiles will
 start in the same DSP sample period.  The HOLD can be set/cleared with a host
 function call, an XMP-Series controller internal variable or a Motor I/O
 state change.

The MEIMotionAttrHold{...} structure is used to configure the HOLD conditions:

 typedef    struct MEIMotionAttrHold {
    MEIMotionAttrHoldType   type;
    MEIMotionAttrHoldSource source;

    float   timeout;
} MEIMotionAttrHold;


The HOLD "type" can be one of the following:

 MEIMotionHoldTypeGATE - host software controlled.
 MEIMotionHoldTypeINPUT - XMP controller internal variable.
 MEIMotionHoldTypeMOTOR - XMP controller motor I/O state change.
```

```
 For each HOLD "type" the "source" union, MEIMotionAttrHoldSource{...},
  configures the HOLD conditions:

  MEIMotionHoldTypeGATE:

   long gate - number between 0 to 31.

   The motion is held until the "gate" is cleared.  The function
   meiControlGateSet(...) is used to set/clear a gate.  When the closed
   parameter is TRUE, the gate is set, and motion is held.  When the
   closed parameter is FALSE, the gate is cleared, and motion starts.

  MEIMotionHoldTypeINPUT

   long *input - address of XMP memory location.
   long mask - bit mask, bitwise ANDed with the input value.
   long pattern - HOLD is released when masked value matches pattern.

   The motion is held until the value of the internal XMP memory location
   (pointed to by *input) bitwise ANDed with the mask matches the pattern.

  MEIMotionHoldTypeMOTOR

   long number - motor "n", XMP's dedicated inputs (Motor[n].IO.DedicaedIN.IO)
   long mask - bit mask, bitwise ANDed with the dedicated inputs.
   long pattern - HOLD is released when masked inputs matches pattern.

   The motion is held until the value of the dedicated input word
   (Motor[n].IO.DedicaedIN.IO) bitwise ANDed with the mask matches the
   pattern.


 The HOLD "timeout" will cause the motion to start after the specified period
  (in seconds) even if the hold criteria have not been met.  To disable the
  timeout feature, set the timeout value to zero.

 The MPI expects an array of hold attributes specifying separate attributes
  form each axis of a motion supervisor. All axes holding with the same hold
  attributes (same gate, same input, mask, and pattern) will start motion in
  the same sample even if the moves are specified using different motion
  supervisors.

 Warning!  This is a sample program to assist in the integration of the
  XMP motion controller with your application.  It may not contain all
  of the logic and safety features that your application requires.
 */

 #include <stdlib.h>
 #include <stdio.h>

 #include "stdmpi.h"
 #include "stdmei.h"
```

```
#include "apputil.h"

#define MOTION_NUMBER   (0)
#define AXIS_NUMBER     (0)
#define GATE_NUMBER     (0)
#define TIMEOUT         (0) /* Wait forever */

#define GOAL_POSITION   ( 10000.0)
#define VELOCITY        ( 10000.0)
#define ACCELERATION    (100000.0)
#define DECELERATION    (100000.0)
#define JERK_PERCENT    (   100.0)



/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber)
{
    long returnValue;

    /* Create control object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(*control);
    msgCHECK(returnValue);
```

```
    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create Motion object */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        *axis);
    msgCHECK(mpiMotionValidate(*motion));
}


/* Delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIMotion   *motion,
                    MPIAxis     *axis)
{
    long returnValue;

    /* Delete motion supervisor object */
    returnValue = mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    /* Delete axis object */
    returnValue = mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete control object */
    returnValue = mpiControlDelete(*control);
    msgCHECK(returnValue);
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;        /* motion controller object handle */
    MPIMotion           motion;         /* motion object handle */
    MPIAxis             axis;           /* axis object handle */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotionParams     params;         /* motion parameters */
    MPITrajectory       trajectory;     /* motion trajectory */
    MEIMotionAttributes attributes;     /* motion attributes */
    MEIMotionAttrHold   hold;           /* hold attribute configuration */

    long                gateNumber  = GATE_NUMBER;
    float               gateTimeout = (float)TIMEOUT;

    double              position;       /* final target position */

    long                returnValue;    /* return value from library */
```

```
    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc, argv, &controlType, &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                &axis,
                AXIS_NUMBER);

    /* Set up motion parameters */
    position                = GOAL_POSITION;    /* counts               */
    trajectory.velocity     = VELOCITY;         /* counts per sec       */
    trajectory.acceleration = ACCELERATION;     /* counts per sec * sec */
    trajectory.deceleration = DECELERATION;     /* counts per sec * sec */
    trajectory.jerkPercent  = JERK_PERCENT;

    params.sCurve.trajectory    = &trajectory;
    params.sCurve.position      = &position;

    /* Configure Motion Attributes */
    hold.type = MEIMotionAttrHoldTypeGATE;  /* software motion Gate control */
    hold.source.gate = gateNumber;
    hold.timeout = gateTimeout;

    attributes.hold = &hold;
    params.external = &attributes;

    printf("Setting the Control Gate to TRUE.\n");

    /* Set a control gate to prevent the motion profile from executing */
    returnValue =
        meiControlGateSet(control,
                          gateNumber,
                          TRUE);    /* set gate */
    msgCHECK(returnValue);

    printf("Loading the motion.\n");

    /* Load motion profile with HOLD attribute */
    returnValue =
        mpiMotionStart(motion,
                       (MPIMotionType)
                            (MPIMotionTypeS_CURVE | MEIMotionAttrMaskHOLD),
                       &params);
    msgCHECK(returnValue);

    printf("Press any key to set the Control Gate to FALSE...\r");

    /* Wait for user key */
    meiPlatformKey(MPIWaitFOREVER);
```

```
    printf("Setting the Control Gate to FALSE (executing the motion).\n");

    /* Clear a control gate, executing the preloaded motion */
    returnValue =
        meiControlGateSet(control,
                          gateNumber,
                          FALSE);   /* clear the gate */
    msgCHECK(returnValue);

    /* Delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis);

    return ((int)returnValue);
}
```

motGate2.c -- Load and trigger motion profiles using a control Gate.

```
/* motGate2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/motgate2.c 7     8/01/01 2:08p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    motGate2Main
argMainRENAME(main, motGate2)
#endif

/*

:Load and trigger motion profiles using a control Gate.

This sample program demonstrates how to use the motion HOLD attribute and
 a control Gate, to pre-load a single axis motion, and to synchronize the
 start of two independent motion profiles.

Here are the steps:

 1) Move the X axis.
 2) Set a control Gate.
 3) Load a Y axis motion profile, using the HOLD attribute.
 4) Clear the control Gate, causing the Y axis motion to execute.
 5) Set a control Gate.
 6) Load X and Y axis motion profiles, using the HOLD attribute.
 7) Clear the control Gate, causing the X and Y axis motions to execute
    simultaneously.


The XMP-Series controller supports a motion HOLD attribute, which is useful
 for preloading and triggering motion profiles.  One or more motion supervisors
 can be started from the same HOLD conditions.  When multiple motion supervisors
 are triggered by the same HOLD conditions, the individual motion profiles will
 start in the same DSP sample period.  The HOLD can be set/cleared with a host
 function call, an XMP-Series controller internal variable or a Motor I/O
 state change.

The MEIMotionAttrHold{...} structure is used to configure the HOLD conditions:
```

```
 typedef    struct MEIMotionAttrHold {
    MEIMotionAttrHoldType   type;
    MEIMotionAttrHoldSource source;

    float   timeout;
} MEIMotionAttrHold;
```

The HOLD "type" can be one of the following:

```
 MEIMotionHoldTypeGATE - host software controlled.
 MEIMotionHoldTypeINPUT - XMP controller internal variable.
 MEIMotionHoldTypeMOTOR - XMP controller motor I/O state change.
```

For each HOLD "type" the "source" union, MEIMotionAttrHoldSource{...}, configures the HOLD conditions:

```
 MEIMotionHoldTypeGATE:

  long gate - number between 0 to 31.

  The motion is held until the "gate" is cleared.  The function
  meiControlGateSet(...) is used to set/clear a gate.  When the closed
  parameter is TRUE, the gate is set, and motion is held.  When the
  closed parameter is FALSE, the gate is cleared, and motion starts.

 MEIMotionHoldTypeINPUT

  long *input - address of XMP memory location.
  long mask - bit mask, bitwise ANDed with the input value.
  long pattern - HOLD is released when masked value matches pattern.

  The motion is held until the value of the internal XMP memory location
  (pointed to by *input) bitwise ANDed with the mask matches the pattern.

 MEIMotionHoldTypeMOTOR

  long number - motor "n", XMP's dedicated inputs (Motor[n].IO.DedicaedIN.IO)
  long mask - bit mask, bitwise ANDed with the dedicated inputs.
  long pattern - HOLD is released when masked inputs matches pattern.

  The motion is held until the value of the dedicated input word
  (Motor[n].IO.DedicaedIN.IO) bitwise ANDed with the mask matches the
  pattern.
```

The HOLD "timeout" will cause the motion to start after the specified period (in seconds) even if the hold criteria have not been met. To disable the timeout feature, set the timeout value to zero.

The MPI expects an array of hold attributes specifying separate attributes form each axis of a motion supervisor. All axes holding with the same hold attributes (same gate, same input, mask, and pattern) will start motion in the same sample even if the moves are specified using different motion

```
 supervisors.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define MOTION_NUMBER1  (0)
#define MOTION_NUMBER2  (1)
#define AXIS_NUMBER1    (0)
#define AXIS_NUMBER2    (1)
#define GATE_NUMBER     (0)
#define TIMEOUT         (0) /* Wait forever */

#define GOAL_POSITION1  ( 10000.0)
#define GOAL_POSITION2  (     0.0)
#define VELOCITY        ( 10000.0)
#define ACCELERATION    (100000.0)
#define DECELERATION    (100000.0)
#define JERK_PERCENT    (   100.0)


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motionX,
```

```
                    MPIMotion          *motionY,
                    long               *motionNumber,
                    MPIAxis            *axisX,
                    MPIAxis            *axisY,
                    long               *axisNumber)
{
    long returnValue;

    /* Create control object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create X axis object using AXIS_X number on controller */
    *axisX =
        mpiAxisCreate(*control,
                      axisNumber[0]);       /* axis Number */
    msgCHECK(mpiAxisValidate(*axisX));

    /* Create Y axis object using AXIS_Y number on controller */
    *axisY =
        mpiAxisCreate(*control,
                      axisNumber[1]);       /* axis Number */
    msgCHECK(mpiAxisValidate(*axisY));

    /* Create Motion object for axisX */
    *motionX =
        mpiMotionCreate(*control,
                        motionNumber[0],        /* motion supervisor number */
                        *axisX);
    msgCHECK(mpiMotionValidate(*motionX));

    /* Create Motion object for axisY */
    *motionY =
        mpiMotionCreate(*control,
                        motionNumber[1],        /* motion supervisor number */
                        *axisY);
    msgCHECK(mpiMotionValidate(*motionY));
}


/* Delete MPI objects */
void programCleanup(MPIControl  *control,
                    MPIMotion   *motionX,
                    MPIMotion   *motionY,
                    MPIAxis     *axisX,
                    MPIAxis     *axisY)
{
    long returnValue;
```

```
    /* Delete motion supervisor Y */
    returnValue = mpiMotionDelete(*motionY);
    msgCHECK(returnValue);

    /* Delete motion supervisor X */
    returnValue = mpiMotionDelete(*motionX);
    msgCHECK(returnValue);

    /* Delete axis Y */
    returnValue = mpiAxisDelete(*axisY);
    msgCHECK(returnValue);

    /* Delete axis X */
    returnValue = mpiAxisDelete(*axisX);
    msgCHECK(returnValue);

    /* Delete control object */
    returnValue = mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/*
  waitForMotionDone() polls motion to see if the axis is in an idle or error
  state every 10ms. When it gets an MPIStateIdle, MPIStateERROR, or
  MPIStateSTOPPING_ERROR then the waitForMotionDone() exits.
*/
void waitForMotionDone(MPIMotion motion)
{
    MPIStatus   status;
    long        motionDone;
    long        returnValue = 0;


    /* Poll status until motion done */
    motionDone = FALSE;
    while (motionDone == FALSE) {

        /* Get the motion supervisor status */
        returnValue =
            mpiMotionStatus(motion,
                            &status,
                            NULL);
        msgCHECK(returnValue);

        switch (status.state) {
            case MPIStateSTOPPING:
            case MPIStateMOVING: {
                /* Sleep for 10ms and give up control to other threads */
                meiPlatformSleep(10);
                break;
            }
            case MPIStateIDLE:
            case MPIStateERROR:
            case MPIStateSTOPPING_ERROR: {
```

```
                /* Motion is done */
                motionDone = TRUE;
                break;
            }
            default: {
                /* Unknown State */
                fprintf(stderr, "Unknown state from mpiMotionStatus.\n");
                msgCHECK(MPIMessageFATAL_ERROR);
                break;
            }
        }
    }
}


/* Function demonstrating how gates can control motion */
void gateControlledMotion(MPIControl         control,
                          MPIMotion          motion1,
                          MPIMotion          motion2,
                          MPIMotionParams   *motionParams,
                          MPIMotionType      motionType,
                          long               gateNumber,
                          long               hold)
{
    long returnValue;


    /* If hold = TRUE, turn on the HOLD bit of MPIMotionType */
    if (hold==TRUE) {
        motionType = (MPIMotionType) (motionType | MEIMotionAttrMaskHOLD);
    }

    printf("Turning control gate ON.\r");

    /* Set a control gate to prevent the motion profile from executing */
    returnValue =
        meiControlGateSet(control,
                          gateNumber,
                          TRUE);    /* set gate */
    msgCHECK(returnValue);

    printf("Turned control gate ON. \n");
    printf("Press any key to call mpiMotionStart(...) %s HOLD attribute.\r",
           (hold) ? "with" : "without");

    /* Wait for a key form the user */
    meiPlatformKey(MPIWaitFOREVER);

    /* Load motion profile */
    returnValue = mpiMotionStart(motion1,
                                 motionType,
                                 motionParams);
    msgCHECK(returnValue);

    /* If user specified a second motion object */
```

```
    if (motion2!=MPIHandleVOID) {

        /* Load motion profile */
        returnValue = mpiMotionStart(motion2,
                                     motionType,
                                     motionParams);
        msgCHECK(returnValue);
    }

    printf("Called mpiMotionStart(...) %s HOLD attribute.              \n",
           (hold) ? "with" : "without");

    printf("Press any key to turn control gate OFF\r");

    /* Wait for a key form the user */
    meiPlatformKey(MPIWaitFOREVER);

    /* Clear a control gate, executing the preloaded motion */
    returnValue =
        meiControlGateSet(control,
                          gateNumber,
                          FALSE);   /* clear the gate */
    msgCHECK(returnValue);

    printf("Turned control gate OFF.              \n");
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;    /* motion controller handle */
    MPIAxis             axisX;      /* X axis                   */
    MPIAxis             axisY;      /* Y axis                   */
    MPIMotion           motionX;    /* motion object for axisX  */
    MPIMotion           motionY;    /* motion object for axisY  */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotionParams     motionParams;   /* motion parameters            */
    MPITrajectory       trajectory;     /* trajectory information       */
    MEIMotionAttributes attributes;     /* motion attributes            */
    MEIMotionAttrHold   hold;           /* hold attribute configuration */

    long    axisNumber[2]   = { AXIS_NUMBER1,   AXIS_NUMBER2,   };
    long    motionNumber[2] = { MOTION_NUMBER1, MOTION_NUMBER2, };
    long    gateNumber      = GATE_NUMBER;
    float   gateTimeout     = (float)TIMEOUT;
    double  position;


    /* Set up motion parameters */
    trajectory.velocity     = VELOCITY;         /* counts per sec       */
    trajectory.acceleration = ACCELERATION;     /* counts per sec * sec */
    trajectory.deceleration = DECELERATION;     /* counts per sec * sec */
    trajectory.jerkPercent  = JERK_PERCENT;
```

```
    motionParams.sCurve.trajectory  = &trajectory;
    motionParams.sCurve.position    = &position;

    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc, argv, &controlType, &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motionX,
                &motionY,
                motionNumber,
                &axisX,
                &axisY,
                axisNumber);

    /* Configure Motion Attributes */
    hold.type = MEIMotionAttrHoldTypeGATE;  /* software motion Gate control */
    hold.source.gate = gateNumber;
    hold.timeout = gateTimeout;

    attributes.hold = &hold;
    motionParams.external = &attributes;


    /* Tell axis to go to GOAL_POSITION1 */
    position = GOAL_POSITION1;  /* counts */

    printf("\n*** X AXIS ***\n");

    /* Execute X-axis motion without HOLD attribute */
    gateControlledMotion(control,
                         motionX,
                         MPIHandleVOID,
                         &motionParams,
                         MPIMotionTypeS_CURVE,
                         gateNumber,
                         FALSE);

    printf("Wating for motion to complete...\n");

    /* Wait for motion to complete */
    waitForMotionDone(motionX);

    printf("\n*** Y AXIS ***\n");

    /* Execute Y-axis motion with HOLD attribute */
    gateControlledMotion(control,
                         motionY,
                         MPIHandleVOID,
                         &motionParams,
                         MPIMotionTypeS_CURVE,
                         gateNumber,
```

```
                        TRUE);

    printf("Wating for motion to complete...\n");

    /* Wait for motion to complete */
    waitForMotionDone(motionY);


    /* Tell axis to go to GOAL_POSITION2 */
    position = GOAL_POSITION2;  /* counts */

    printf("\n*** X & Y AXES ***\n");

    /* Execute X,Y-axes motion with HOLD attribute */
    gateControlledMotion(control,
                         motionX,
                         motionY,
                         &motionParams,
                         MPIMotionTypeS_CURVE,
                         gateNumber,
                         TRUE);

    /* Delete MPI objects */
    programCleanup(&control,
                   &motionX,
                   &motionY,
                   &axisX,
                   &axisY);

    return MPIMessageOK;
}
```

motid1.c -- Point to Point motion with Motion Done event identification.

```
/* motId1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/motid1.c 19    7/23/01 2:36p Kevinh $";
#endif

/*

:Point to Point motion with Motion Done event identification.

The MPI supports a motion attribute that allows an application to tag
 a mpiMotionStart(...) or mpiMotionModify(...) with an identification
 value.

This feature is useful in tracking the commanded motions to the done
 events.  It is also applicable when using the AUTO_START attribute
 with mpiMotionModify(...).

For example:

Suppose mpiMotionStart(...) is called with the MPIMotionAttrID attribute,
 and the params.attributes.id = 1 and later mpiMotionModify(...) is called
 with params.attributes.id = 2.

Case 1: MPIMotionAttrAUTO_START is not used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageIDLE, there will be 1 DONE
 event with info->data.motion.id = 1.

Case 2: MPIMotionAttrAUTO_START is used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageAUTO_START,
 there will be 2 DONE events. The first will have info->data.motion.id = 1
 and the second will have info->data.motion.id = 2.


The XMP-Series controller supports some user data to be copied during Events
 and then later retrieved from the Host via the Notify object.

There are MEIXmpSignalUserData words of data for the motion, axis and motor
 event sources.  The default configuration for the first word is the XMP's
```

```
 sampleCounter.  The default for the second word is "actualPosition" for axis
 event sources, and "encoderPosition" for motor event sources.

In this sample, the second word for motion and axis events is configured for "id".

Note:  Keeping the first word configured for the "sampleCounter" is very
 helpful in determining event sequences.

Here is the default configuration for the event data:

    union {
        long sampleCounter;
        struct {
            long sampleCounter;
        } motion;
        struct {
            long sampleCounter;
            long actualPosition;
        } axis;
        struct {
            long sampleCounter;
            long encoderPosition;
        } motor;
        long word[MEIXmpSignalUserData];
    } data;


Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motId1Main

argMainRENAME(main, motId1)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (1)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0,   };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeS_CURVE;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
```

```
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Motion Parameters */
double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  },
    { 0.0,      },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent*/
        { 10000.0,      100000.0,   100000.0,   0.0,    },
    },
};

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    MEIEventNotifyData  motionData;
    MEIXmpAxis          *xmpAxis;

    long    returnValue;    /* return value from library */

    long    index;
    long    moveId;         /* identification tag */
```

```
    long    motionDone;

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
```

```
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object for axisNumber */
    axis =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motion object, appending the axis object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Get current user data configuration */
    mpiEventMaskCLEAR(eventMask);
    returnValue =
        mpiMotionEventNotifyGet(motion,
                                &eventMask,
                                &motionData);
    msgCHECK(returnValue);

    /* Request notification of all events from motion */
    mpiEventMaskALL(eventMask);

    /* Request ID data from motion event info[...] */
    returnValue  =
        mpiAxisMemory(axis,     /* use ID from first axis in motion object */
                      &(void *)xmpAxis);
    msgCHECK(returnValue);

    motionData.address[1] = (void *)(&xmpAxis->MoveID);

    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                &motionData);
    msgCHECK(returnValue);
```

```
    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    printf("Press any key to stop ...\n");

    /* Loop repeatedly */
    index       = 0;
    moveId      = 0;
    motionDone  = TRUE;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus  eventStatus;

        if (motionDone == TRUE) {
            MPIMotionParams motionParams;       /* motion parameters */

            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }

            motionParams.attributes.id = moveId;        /* identification tag */
```

```
            returnValue =
                mpiMotionStart(motion,
                               (MPIMotionType)(motionType | MPIMotionAttrMaskID),
                               &motionParams);
            msgCHECK(returnValue);

            printf("\nMotion Start #%d...", moveId);

            moveId++;

            motionDone = FALSE;
        }

        /* Wait for motion event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);
        msgCHECK(returnValue);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            MEIEventStatusInfo *info;

            info = (MEIEventStatusInfo *)eventStatus.info;

            printf(" done #%ld",
                    info->data.word[1]);        /* user configurable data */

            if (++index >= MOTION_COUNT) {
                index = 0;
            }

            motionDone = TRUE;
        }
    }

    printf("\n");

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);
```

```
    return ((int)returnValue);
}
```

motid2.c -- Point to Point motion with motion/axis event identification.

```
/* motId2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Point to Point motion with motion/axis event identification.

The MPI supports a motion attribute that allows an application to tag
 a mpiMotionStart(...) or mpiMotionModify(...) with an identification
 value.

This feature is useful in tracking the commanded motions to the done
 events.  It is also applicable when using the AUTO_START attribute
 with mpiMotionModify(...).

For example:

Suppose mpiMotionStart(...) is called with the MPIMotionAttrID attribute,
 and the params.attributes.id = 1 and later mpiMotionModify(...) is called
 with params.attributes.id = 2.

Case 1: MPIMotionAttrAUTO_START is not used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageIDLE, there will be 1 DONE
 event with info->data.motion.id = 1.

Case 2: MPIMotionAttrAUTO_START is used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageAUTO_START,
 there will be 2 DONE events. The first will have info->data.motion.id = 1
 and the second will have info->data.motion.id = 2.


The XMP-Series controller supports some user data to be copied during Events
 and then later retrieved from the Host via the Notify object.

There are MEIXmpSignalUserData words of data for the motion, axis and motor
 event sources.  The default configuration for the first word is the XMP's
 sampleCounter.  The default for the second word is "actualPosition" for axis
 event sources, and "encoderPosition" for motor event sources.

In this sample, the second word for motion and axis events is configured for "id".
```

```
 Note:  Keeping the first word configured for the "sampleCounter" is very
  helpful in determining event sequences.

 Here is the default configuration for the event data:

     union {
         long sampleCounter;
         struct {
             long sampleCounter;
         } motion;
         struct {
             long sampleCounter;
             long actualPosition;
         } axis;
         struct {
             long sampleCounter;
             long encoderPosition;
         } motor;
         long word[MEIXmpSignalUserData];
     } data;


 Warning!  This is a sample program to assist in the integration of the
  XMP motion controller with your application.  It may not contain all
  of the logic and safety features that your application requires.
 */

 #include <stdlib.h>
 #include <stdio.h>

 #include "stdmpi.h"
 #include "stdmei.h"

 #include "apputil.h"

 #if defined(ARG_MAIN_RENAME)
 #define main    motId2Main

 argMainRENAME(main, motId2)
 #endif

 #define MOTION_COUNT    (2)
 #define AXIS_COUNT      (1)

 /* Command line arguments and defaults */
 long            axisNumber[AXIS_COUNT] = { 0,   };
 long            motionNumber    = 0;
 MPIMotionType   motionType      = MPIMotionTypeS_CURVE;

 Arg argList[] = {
     {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
     {   "-motion",  ArgTypeLONG,    &motionNumber,  },
     {   "-type",    ArgTypeLONG,    &motionType,    },

     {   NULL,       ArgTypeINVALID, NULL,   }
 };
```

```
/* Motion Parameters */
double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  },
    { 0.0,      },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent*/
        { 10000.0,      100000.0,   100000.0,   0.0,    },
    },
};

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    MEIEventNotifyData  motionData;
    MEIXmpAxis          *xmpAxis;

    long    returnValue;    /* return value from library */

    long    index;
    long    moveId;     /* identification tag */
    long    motionDone; /* flag when Done occurs */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
```

```
    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
```

```
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object for axisNumber */
    axis =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motion object, appending the axis object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Get current user data configuration */
    mpiEventMaskCLEAR(eventMask);
    returnValue =
        mpiMotionEventNotifyGet(motion,
                                &eventMask,
                                &motionData);
    msgCHECK(returnValue);

    /* Request notification of ALL events from motion */
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);

    /* Request ID data from motion event info[...] */
    returnValue  =
        mpiAxisMemory(axis,     /* use ID from first axis in motion object */
                      &(void *)xmpAxis);
    msgCHECK(returnValue);

    motionData.address[1] = (void *)(&xmpAxis->MoveID);

    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                &motionData);
    msgCHECK(returnValue);

    /* Configure sampleCounter and ID to be returned from Axis event sources */
    returnValue =
        mpiAxisEventNotifySet(axis,
                              eventMask,    /* use the same eventMask as motion */
                              &motionData); /* use the same data as motion */
```

```
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    printf("Press any key to stop ...\n");

    /* Loop repeatedly */
    index       = 0;
    moveId      = 0;
    motionDone  = TRUE;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus      eventStatus;
        MEIEventStatusInfo  *info;

        if (motionDone) {
            MPIMotionParams     motionParams;       /* motion parameters */

            /* fill in the MPIMotionParams structure */
            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
```

```
        }

        motionParams.attributes.id = moveId;    /* identification tag */

        returnValue =
            mpiMotionStart(motion,
                           (MPIMotionType)(motionType | MPIMotionAttrMaskID),
                           &motionParams);
        msgCHECK(returnValue);

        printf("\n\nMotion Start #%d...\n", moveId);

        moveId++;

        motionDone = FALSE;
    }

    /* Wait for motion event */
    returnValue =
        mpiNotifyEventWait(notify,
                           &eventStatus,
                           MPIWaitFOREVER);
    msgCHECK(returnValue);

    info = (MEIEventStatusInfo *)eventStatus.info;

    switch (eventStatus.type) {
        /* In Coarse Event from axis source */
        case MEIEventTypeIN_POSITION_COARSE: {
            printf("  InCoarse #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeIN_POSITION_FINE: {
            printf("  InFine #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeAT_TARGET: {
            printf("  AtTarget #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.axis.sampleCounter);
            break;
        }
        /* Motion Done Event from motion source */
        case MPIEventTypeMOTION_DONE: {
            printf("  Done #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.motion.sampleCounter);

            motionDone = TRUE;
```

```
                if (++index >= MOTION_COUNT) {
                    index = 0;
                }
                break;
            }
            default: {
                printf("  %d #%ld(%ld)",
                       eventStatus.type,
                       info->data.word[1],
                       info->data.word[0]);
                break;
            }
        }
    }

    printf("\n");

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

motid3.c -- Point to Point motion with ID, APPEND, and HOLD attributes.

```
/* motId3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Point to Point motion with ID, APPEND, and HOLD attributes.

The MPI supports a motion attribute that allows an application to tag
 a mpiMotionStart(...) or mpiMotionModify(...) with an identification
 value.

This feature is useful in tracking the commanded motions to the done
 events.  It is also applicable when using the AUTO_START attribute
 with mpiMotionModify(...).

For example:

Suppose mpiMotionStart(...) is called with the MPIMotionAttrID attribute,
 and the params.attributes.id = 1 and later mpiMotionModify(...) is called
 with params.attributes.id = 2.

Case 1: MPIMotionAttrAUTO_START is not used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageIDLE, there will be 1 DONE
 event with info->data.motion.id = 1.

Case 2: MPIMotionAttrAUTO_START is used.
if mpiMotionModify(...) returns MPIMessageOK, there will be 1 DONE event
 with info->data.motion.id = 2.
if mpiMotionModify(...) returns MPIMotionMessageAUTO_START,
 there will be 2 DONE events. The first will have info->data.motion.id = 1
 and the second will have info->data.motion.id = 2.


The XMP-Series controller supports some user data to be copied during Events
 and then later retrieved from the Host via the Notify object.

There are MEIXmpSignalUserData words of data for the motion, axis and motor
 event sources.  The default configuration for the first word is the XMP's
 sampleCounter.  The default for the second word is "actualPosition" for axis
 event sources, and "encoderPosition" for motor event sources.

In this sample, the second word for motion and axis events is configured for "id".
```

Note: Keeping the first word configured for the "sampleCounter" is very helpful in determining event sequences.

Here is the default configuration for the event data:

```
    union {
        long sampleCounter;
        struct {
            long sampleCounter;
        } motion;
        struct {
            long sampleCounter;
            long actualPosition;
        } axis;
        struct {
            long sampleCounter;
            long encoderPosition;
        } motor;
        long word[MEIXmpSignalUserData];
    } data;
```

The APPEND attribute is useful for buffering moves in the controller. When the APPEND attribute is used with mpiMotionStart(...), a new motion profile is added to the end of any previously loaded motion profiles in the controller. The XMP-Series controller automatically buffers the APPENDed MotionStarts, executing each sequentially after the Motion Done criteria has been met for each profile. Each MotionStart profile generates a Motion Done event to the Host.

When APPEND attribute is used with mpiMotionModify(...), the motion profile is added to the end of any previously loaded motion profiles in the controller. The XMP-Series controller automatically buffers the APPENDed MotionModifies, executing each sequentially. Only the very last MotionModify waits for the Motion Done criteria to be met before generating a Motion Done event to the host.

The XMP-Series controller supports a motion HOLD attribute, which is useful for preloading and triggering motion profiles. One or more motion supervisors can be started from the same HOLD conditions. When multiple motion supervisors are triggered by the same HOLD conditions, the individual motion profiles will start in the same DSP sample period. The HOLD can be set/cleared with a host function call, an XMP-Series controller internal variable or a Motor I/O state change.

The MEIMotionAttrHold{...} structure is used to configure the HOLD conditions:

```
 typedef    struct MEIMotionAttrHold {
    MEIMotionAttrHoldType   type;
    MEIMotionAttrHoldSource source;

    float   timeout;
} MEIMotionAttrHold;
```

The HOLD "type" can be one of the following:

```
 MEIMotionHoldTypeGATE - host software controlled.
 MEIMotionHoldTypeINPUT - XMP controller internal variable.
 MEIMotionHoldTypeMOTOR - XMP controller motor I/O state change.


For each HOLD "type" the "source" union, MEIMotionAttrHoldSource{...},
 configures the HOLD conditions:

 MEIMotionHoldTypeGATE:

  long gate - number between 0 to 31.

  The motion is held until the "gate" is cleared.  The function
  meiControlGateSet(...) is used to set/clear a gate.  When the closed
  parameter is TRUE, the gate is set, and motion is held.  When the
  closed parameter is FALSE, the gate is cleared, and motion starts.

 MEIMotionHoldTypeINPUT

  long *input - address of XMP memory location.
  long mask - bit mask, bitwise ANDed with the input value.
  long pattern - HOLD is released when masked value matches pattern.

  The motion is held until the value of the internal XMP memory location
  (pointed to by *input) bitwise ANDed with the mask matches the pattern.

 MEIMotionHoldTypeMOTOR

  long number - motor "n", XMP's dedicated inputs (Motor[n].IO.DedicaedIN.IO)
  long mask - bit mask, bitwise ANDed with the dedicated inputs.
  long pattern - HOLD is released when masked inputs matches pattern.

  The motion is held until the value of the dedicated input word
  (Motor[n].IO.DedicaedIN.IO) bitwise ANDed with the mask matches the
  pattern.


The HOLD "timeout" will cause the motion to start after the specified period
 (in seconds) even if the hold criteria have not been met.  To disable the
 timeout feature, set the timeout value to zero.

The MPI expects an array of hold attributes specifying separate attributes
 form each axis of a motion supervisor. All axes holding with the same hold
 attributes (same gate, same input, mask, and pattern) will start motion in
 the same sample even if the moves are specified using different motion
 supervisors.


Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>
```

```
#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motionId3Main

argMainRENAME(main, motionId3)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (1)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0,   };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeS_CURVE;
long            gateNumber = 0;
float           gateTimeout = (float)0.0;


Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },
    {   "-gate",    ArgTypeLONG,    &gateNumber,    },
    {   "-timeout", ArgTypeFLOAT,   &gateTimeout,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Motion Parameters */
double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  },
    { 0.0,      },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent*/
        { 10000.0,      100000.0,   100000.0,   0.0,    },
    },
};

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
```

```
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    MEIEventNotifyData  motionData;
    MEIXmpAxis          *xmpAxis;

    MPIMotionParams     motionParams;   /* motion parameters */
    MEIMotionAttributes attributes;     /* motion attributes */
    MEIMotionAttrHold   hold;           /* hold attribute configuration */

    long    returnValue;    /* return value from library */

    long    index;
    long    moveId;     /* identification tag */
    long    motionDone; /* flag when Done occurs */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
```

```
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST) ||
        (gateNumber < 0) ||
        (gateNumber >= MEIXmpMaxGates) ||
        (gateTimeout < 0.0)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n"
                           "\t\t[-gate # (0 .. %d)]\n"
                           "\t\t[-timeout # (0.0 ..)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1,
                            MEIXmpMaxGates - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object for axisNumber */
    axis =
```

```
    mpiAxisCreate(control,
                  axisNumber[0]);
msgCHECK(mpiAxisValidate(axis));

/* Create motion object, appending the axis object */
motion =
    mpiMotionCreate(control,
                    motionNumber,
                    axis);
msgCHECK(mpiMotionValidate(motion));

/* Get current user data configuration */
mpiEventMaskCLEAR(eventMask);
returnValue =
    mpiMotionEventNotifyGet(motion,
                            &eventMask,
                            &motionData);
msgCHECK(returnValue);

/* Request notification of ALL events from motion */
mpiEventMaskALL(eventMask);
meiEventMaskALL(eventMask);

/* Request ID data from motion event info[...] */
returnValue  =
    mpiAxisMemory(axis,     /* use ID from first axis in motion object */
                  &(void *)xmpAxis);
msgCHECK(returnValue);

/* Configure the second user data word for the ID */
motionData.address[1] = (void *)(&xmpAxis->MoveID);

/* Configure sampleCounter and ID to be returned from Motion event sources */
returnValue =
    mpiMotionEventNotifySet(motion,
                            eventMask,
                            &motionData);
msgCHECK(returnValue);

/* Configure sampleCounter and ID to be returned from Axis event sources */
returnValue =
    mpiAxisEventNotifySet(axis,
                          eventMask,    /* use the same eventMask as motion */
                          &motionData); /* use the same data as motion */
msgCHECK(returnValue);

/* Create event notification object for motion */
notify =
    mpiNotifyCreate(eventMask,
                    motion);
msgCHECK(mpiNotifyValidate(notify));

/* Create event manager object */
eventMgr = mpiEventMgrCreate(control);
msgCHECK(mpiEventMgrValidate(eventMgr));
```

```
    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Configure Motion Attributes */
    hold.type = MEIMotionAttrHoldTypeGATE;  /* software motion Gate control */
    hold.source.gate = gateNumber;
    hold.timeout = gateTimeout;

    attributes.hold = &hold;
    motionParams.external = &attributes;


    printf("Press any key to stop ...\n");

    /* Loop repeatedly */
    index       = 0;
    moveId      = 0;
    motionDone  = TRUE;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus      eventStatus;
        MEIEventStatusInfo  *info;

        if (motionDone) {

            /* Set a control gate to prevent motion profile from executing */
            returnValue =
                meiControlGateSet(control,
                                  gateNumber,
                                  TRUE);    /* set gate */
            msgCHECK(returnValue);

            for (index = 0; index < MOTION_COUNT; index++) {
                /* fill in the MPIMotionParams structure */
                switch (motionType) {
                    case MPIMotionTypeS_CURVE: {
                        motionParams.sCurve = sCurve[index];
                        break;
                    }
                    case MPIMotionTypeTRAPEZOIDAL: {
                        motionParams.trapezoidal = trapezoidal[index];
                        break;
                    }
                    case MPIMotionTypeVELOCITY: {
                        motionParams.velocity = velocity[index];
                        break;
                    }
```

```
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }

            motionParams.attributes.id = moveId;    /* identification tag */

            returnValue =
                mpiMotionStart(motion,
                               (MPIMotionType)(motionType |
                               MPIMotionAttrMaskID |
                               MPIMotionAttrMaskAPPEND |
                               MEIMotionAttrMaskHOLD),
                               &motionParams);
            msgCHECK(returnValue);

            printf("\nMotion Start #%d...\n", moveId);

            moveId++;
        }

        /* Clear a control gate, executing the preloaded motion */
        returnValue =
            meiControlGateSet(control,
                              gateNumber,
                              FALSE);   /* clear the gate */
        msgCHECK(returnValue);

        index = 0;
        motionDone = FALSE;
    }

    /* Wait for motion event */
    returnValue =
        mpiNotifyEventWait(notify,
                           &eventStatus,
                           MPIWaitFOREVER);
    msgCHECK(returnValue);

    info = (MEIEventStatusInfo *)eventStatus.info;

    switch (eventStatus.type) {
        /* In Coarse Event from axis source */
        case MEIEventTypeIN_POSITION_COARSE: {
            printf("  InCoarse #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeIN_POSITION_FINE: {
            printf("  InFine #%ld(%ld)",
                   info->data.word[1],      /* user configurable data */
                   info->data.axis.sampleCounter);
            break;
```

```
            }
            /* In Fine Event from axis source */
            case MEIEventTypeAT_TARGET: {
                printf("  AtTarget #%ld(%ld)",
                       info->data.word[1],      /* user configurable data */
                       info->data.axis.sampleCounter);
                break;
            }
            /* Motion Done Event from motion source */
            case MPIEventTypeMOTION_DONE: {
                printf("  Done #%ld(%ld)\n",
                       info->data.word[1],
                       info->data.motion.sampleCounter);

                if (++index >= MOTION_COUNT) {
                    index = 0;
                    motionDone = TRUE;
                }
                break;
            }
            default: {
                printf("  %d #%ld(%ld)",
                       eventStatus.type,
                       info->data.word[1],
                       info->data.word[0]);
                break;
            }
        }
    }

    printf("\n");

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

motion1.c -- Perform point to point trapezoidal profile motion.

```
/* motion1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/motion1.c 12    7/18/01 9:32a Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    motion1Main

argMainRENAME(main, motion1)
#endif

/*

:Perform point to point trapezoidal profile motion.

This is a simple program to demonstrate how to start a trapezoidal profile
 motion on a single axis.  There is a minimal error checking in this sample.

Warning!  This is a sample program to assist in the longegration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "stdmpi.h"
#include "stdmei.h"
#include "apputil.h"


#define MOTION_NUMBER   (0)
#define AXIS_NUMBER     (0)

#define GOAL_POSITION   (50000)
#define VELOCITY        (5000)
#define ACCELERATION    (10000)
#define DECELERATION    (20000)
```

```
/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber)
{
    long           returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,       /* motion supervisor number */
                        *axis);             /* axis object handle */
    msgCHECK(mpiMotionValidate(*motion));
```

```
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis)
{
    long    returnValue;


    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    /* Delete axis object */
    returnValue =
        mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Command simple trapezoidal motion */
long simpleTrapMove(MPIMotion   motion,
                    double      goalPosition,
                    double      velocity,
                    double      acceleration,
                    double      deceleration)
{
    MPIMotionParams     params;     /* Motion parameters      */
    MPITrajectory       trajectory; /* Trajectory information */

    long returnValue;               /* MPI library return value */


    /* Setup trajectory structure */
    trajectory.velocity     = velocity;
    trajectory.acceleration = acceleration;
    trajectory.deceleration = deceleration;

    /* Setup parameters structure */
    params.trapezoidal.trajectory   = &trajectory;
    params.trapezoidal.position     = &goalPosition;

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
```

```
                        &params);
    msgCHECK(returnValue);

    return returnValue;
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;    /* motion controller object handle */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;     /* motion object handle */
    MPIAxis             axis;       /* axis object handle */


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                &axis,
                AXIS_NUMBER);

    /* Command simple motion */
    simpleTrapMove(motion,
                   GOAL_POSITION,
                   VELOCITY,
                   ACCELERATION,
                   DECELERATION);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis);

    return MPIMessageOK;
}
```

motion2.c -- Two-axis motion, with synchronized and coordinated S-Curve profiles.

```
/* motion2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/motion2.c 18    7/23/01 2:36p Kevinh $";
#endif

/*

:Two-axis motion, with synchronized and coordinated S-Curve profiles.

This sample demonstrates how to create a two-axis motion system, using
 a single motion object.  Simple point to point motion is commanded
 using S-Curve (or Trapezoidal) profile.

When motion is commanded to point 0, using the MPIMotionAttrMaskSYNC_START
 attribute, the controller starts the motion profiles for both axes at the
 same time.  Each axis uses it's own MPITrajectory, independently reaching
 their target positions at different times.

When motion is commanded to point 1, (no attributes), the controller
 starts and completes the motion profiles for the axes at the same time.
 Each axis shares a single MPITrajectory, by breaking the vector trajectory
 values into component trajectory values based on the ratio of the axis
 distances.  The result is point to point linear coordinated move for the
 axes, specified by a target "position" for each axis and a single vector
 "trajectory" for the axes.

During motion, the motion status is polled from the controller.  When both
 axes complete their motions (state = IDLE), the command and actual positions,
 and motion status information is displayed.

If an error condition occurs during motion, the program clears the error
 using mpiMotionAction(motion, MPIActionRESET).

Note:  When multiple axes are associated with a motion supervisor, the
 controller automatically combines the individual axis and motor status
 into the motion status.  Thus, if a Stop, E-Stop or Abort action occurs
 on one axis, the event will be propogated automatically to the other axes.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
```

```
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motion2Main

argMainRENAME(main, motion2)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeS_CURVE;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Motion Parameters */
double position[MOTION_COUNT][AXIS_COUNT] = {
    { 2000.0,   20000.0,    },
    { 0.0,      0.0,        },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
};

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
```

```
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

long
    motionIdle(MPIMotion    motion,
               MPIStatus    *status);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axisX;      /* X axis */
    MPIAxis     axisY;      /* Y axis */
    MPIMotion   motion;     /* motion object */

    long    returnValue;    /* return value from library */

    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
```

```
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create X axis object using axis number 0 on controller*/
    axisX =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axisX));

    /* Create Y axis object using axis number 1 on controller */
    axisY =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axisY));

    /* Create motion object */
    /* Append X axis to motion */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axisX);
```

```
    msgCHECK(mpiMotionValidate(motion));

    /* Append Y axis to motion */
    returnValue =
        mpiMotionAxisAppend(motion,
                            axisY);
    msgCHECK(returnValue);

    /* Loop repeatedly */
    index = 0;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        long    motionDone;

        if (returnValue == MPIMessageOK) {
            MPIMotionType   type;
            MPIMotionParams motionParams;

            type = motionType;

            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    if (index == 0) {
                        type = (MPIMotionType)(type | MPIMotionAttrMaskSYNC_START);
                    }
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    if (index == 0) {
                        type = (MPIMotionType)(type | MPIMotionAttrMaskSYNC_START);
                    }
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }

            printf("\nMotionStart...");

            returnValue =
                mpiMotionStart(motion,
                               type,
                               &motionParams);

            if (returnValue != MPIMessageOK) {
                printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                       motion,
                       type,
                       &motionParams,
                       returnValue,
```

```
                    mpiMessage(returnValue, NULL));
            }
        }

        /* Poll status until motion done */
        motionDone = FALSE;
        while (motionDone == FALSE) {
            MPIStatus   status;

            returnValue =
                mpiMotionStatus(motion,
                                &status,
                                NULL);
            msgCHECK(returnValue);

            switch (status.state) {
                case MPIStateIDLE: {
                    motionDone = TRUE;

                    motionIdle(motion,
                               &status);

                    /* Wait for the motor to settle */
                    meiPlatformSleep(300);  /* msec */

                    break;
                }
                case MPIStateERROR: {
                    motionDone = TRUE;

                    /* Clear any error condition(s) */
                    returnValue =
                        mpiMotionAction(motion,
                                        MPIActionRESET);
                    msgCHECK(returnValue);

                    /* Wait for reset to take effect */
                    meiPlatformSleep(100);      /* msec */

                    break;
                }
                default: {
                    break;
                }
            }
        }

        if (++index >= MOTION_COUNT) {
            index = 0;
        }
    }

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisY);
    msgCHECK(returnValue);
```

```
    returnValue = mpiAxisDelete(axisX);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long motionIdle(MPIMotion motion,
                MPIStatus *status)
{
    long    returnValue;

    double  actual[AXIS_COUNT];
    double  command[AXIS_COUNT];

    long    index;

    printf("MotionDone: status: state %d action %d eventMask 0x%x\n"
           "\tatTarget %d settled %d %s\n",
            status->state,
            status->action,
            status->eventMask,
            status->atTarget,
            status->settled,
            (status->settled == FALSE)
                    ? "=== NOT SETTLED ==="
                    : "");

    returnValue =
        mpiMotionPositionGet(motion,
                             actual,
                             command);
    msgCHECK(returnValue);

    /* Display axis positions */
    for (index = 0; index < AXIS_COUNT; index++) {
        printf("\taxis[%d]    position: command %11.3lf\tactual %11.3lf\n",
                index,
                command[index],
                actual[index]);
    }

    return (returnValue);
}
```

motion3.c -- Custom point to point S-curve motion with velocity specified using position.

```
/* motion3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Custom point to point S-curve motion with velocity specified using position.

This program demonstrates how to generate a point to point move using a custom
 velociy profile. The motion profile is specified by a structure called
 moveData[...] containing positions, velocities, and jerk percents.
 The motion is commanded using:

  1) mpiMotionStart(...) with the FINAL_VEL attribute, which loads the first
     point and begins motion.

  2) mpiMotionModify(...) with the FINAL_VEL and APPEND attribute, which adds
     the rest of the motion profiles to the end of the first profile.

The resultant motion profile will reach the "midVelocity" at the specified
 "midPosition" and the "endVelocity" at the specified "endPosition."
 MOVE_SEGMENTS is defined by the size of moveData[...].

The following equation was used to determine accelerations and decelerations.

  acceleration = (velocity1^2 - velocity0^2) / (2 * (position1 - position2))

There is a minimal error checking in this sample.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"
#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motion3Main
```

```
argMainRENAME(main, motion3)
#endif

#define AXIS_NUMBER         (0)     /* Axis to be moved */
#define MOTION_NUMBER       (0)     /* Motion supervisor number */
#define JERK_PERCENT        (100.0)

long    axisNumber      = 0;
long    motionNumber    = 0;

struct {
    double  midposition;
    double  endPosition;
    double  midVelocity;
    double  endVelocity;
    double  jerkPercent;

} moveData[] =  {

/*  midposition,    endPosition,    midVel,     endVel, jerkPercent */
    {1200,          3000,           27000.0,    3000.0,     JERK_PERCENT,   },
    {4200,          6000,           15000.0,    3000.0,     JERK_PERCENT,   },
    {8900,          11300,          44000.0,    0.0,        JERK_PERCENT,   },

/*
    {6000,          0,              15000.0,    0.0,        JERK_PERCENT,   },
   Add additional points like the above line
*/

};



Arg argList[] = {
    {   NULL,       ArgTypeINVALID, NULL,   }
};

#define MOVE_SEGMENTS (sizeof(moveData)/sizeof(moveData[0]))

long    parseCommandLine(int                argc,
                         char               *argv[],
                         MPIControlType     *controlType,
                         MPIControlAddress  *controlAddress);

long    createObjects(MPIControl            *control,
                      MPIAxis               *axis,
                      MPIMotion             *motion,
                      MPIControlType        *controlType,
                      MPIControlAddress     *controlAddress,
                      long                  axisNumber);

long    deleteObjects(MPIControl    *control,
                      MPIAxis       *axis,
                      MPIMotion     *motion);
```

```
int main(int    argc,
         char   *argv[])
{

    MPIControl          control;        /* Motion controller object handle */
    MPIMotion           motion;         /* Motion object handle */
    MPIAxis             axis;           /* Axis object handle */
    MPIMotionParams     params;         /* Motion parameters */
    MPITrajectory       trajectory;     /* Motion trajectory */
    MEIMotionAttributes attributes;     /* Motion attributes */
    MPIMotionAttrMask   attrMask;       /* Motion attributes mask */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    returnValue;
    long    segmentCount;
    double  endPosition;

    parseCommandLine(argc,
                     argv,
                     &controlType,
                     &controlAddress);

    createObjects(&control,
                  &axis,
                  &motion,
                  &controlType,
                  &controlAddress,
                  axisNumber);

    segmentCount = 0;
    moveData[-1].endVelocity = 0;
    moveData[-1].endPosition = 0;

    while (segmentCount < MOVE_SEGMENTS){
        /* Setup new motion parameters */
        attrMask =
            MPIMotionAttrMaskAPPEND |
            MEIMotionAttrMaskFINAL_VEL;

        endPosition              =  moveData[segmentCount].endPosition;
        trajectory.velocity      =  moveData[segmentCount].midVelocity;
        attributes.finalVelocity = &moveData[segmentCount].endVelocity;

    /*
       Acceleration = (velocity1^2 - velocity0^2) / (2 * (position1 - position2))
       Point 0 is 'endPosition' of the previous move. Point 1 is 'midposition'.
    */

        trajectory.acceleration  =
            fabs(((pow(moveData[segmentCount].midVelocity, 2) -
            pow(moveData[segmentCount - 1].endVelocity, 2)) /
            (2 * (moveData[segmentCount].midposition -
            moveData[segmentCount - 1].endPosition))));
```

```
    /*
       Acceleration = (velocity1^2 - velocity0^2) / (2 * (position1 - position2))
       Point 0 is 'midposition'. Point 1 is 'endPosition'.
    */

        trajectory.deceleration  =
            fabs(((pow(moveData[segmentCount].endVelocity, 2) -
            pow(moveData[segmentCount].midVelocity, 2)) /
            (2 * (moveData[segmentCount].endPosition -
            moveData[segmentCount].midposition))));

        trajectory.jerkPercent   =  moveData[segmentCount].jerkPercent;

        if (!segmentCount){
            /* First point */
            params.sCurve.position   = &endPosition;
            params.sCurve.trajectory = &trajectory;
            params.external = &attributes;
            /* Start motion */
            returnValue =
                mpiMotionStart(motion,
                               MPIMotionTypeS_CURVE | attrMask,
                               &params);
            msgCHECK(returnValue);
        }
        else{   /* Not first point */
            /* Modify motion */
            returnValue =
                mpiMotionModify(motion,
                                MPIMotionTypeS_CURVE | attrMask,
                                &params);
            msgCHECK(returnValue);
        }
        segmentCount++;
    }

    deleteObjects(&control,
                  &axis,
                  &motion);

    return (returnValue);
}


long    parseCommandLine(int                argc,
                         char               *argv[],
                         MPIControlType     *controlType,
                         MPIControlAddress  *controlAddress)
{
    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
```

```
                    argv,
                    controlType,
                    controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber > (MEIXmpMAX_Axes)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes,
                            MEIXmpMAX_MSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    return 0;
}


long    createObjects(MPIControl            *control,
                      MPIAxis               *axis,
                      MPIMotion             *motion,
                      MPIControlType        *controlType,
                      MPIControlAddress     *controlAddress,
                      long                  axisNumber)
{

    long    returnValue;

    /* Create motion controller object*/
    *control =
        mpiControlCreate(*controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));
```

```
    /* Initialize the motion controller */
    returnValue = mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object*/
    *axis =
        mpiAxisCreate(*control,
                      AXIS_NUMBER);     /* Axis number */
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion object, append axis */
    *motion =
        mpiMotionCreate(*control,
                        MOTION_NUMBER,  /* Motion supervisor number */
                        *axis);         /* Axis object handle */
    msgCHECK(mpiMotionValidate(*motion));

    returnValue =
        mpiMotionAxisListSet(*motion,
                             1,
                             axis);
    msgCHECK(returnValue);

    return returnValue;
}


long    deleteObjects(MPIControl    *control,
                      MPIAxis       *axis,
                      MPIMotion     *motion)
{

    long returnValue;

    /* Object clean-up */
    returnValue = mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(*control);
    msgCHECK(returnValue);

    return returnValue;
}
```

motmap1.c -- Motion object and Motion Supervisor axis map configuration.

```
/* motMap1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/motmap1.c 11    7/23/01 2:36p Kevinh $";
#endif

/*

:Motion object and Motion Supervisor axis map configuration.

The MPI supports Motion objects and Axis objects which reside in the
 host computer.  Several Motion objects can be created and associated
 with single or multiple Axis objects.  Additionally, several Motion
 objects can be configured to share one or more Axis objects.

The XMP controller has Motion Supervisors and Axes.  The controller's
 Motion Supervisors can be associated with a single axis or multiple axes.
 Additionally, several Motion Supervisors can be configured to share
 one or more axes.

When multiple axes are configured for a Motion Supervisor, the controller
 automatically manages the status and error conditions.  For example, if
 a Motion Supervisor is configured with two axes, and one of the axes faults
 with a position error limit, causing an Abort Event, the Motion Supervisor
 will generate Abort Event on the other axis.

Some care must be used with Motion Supervisor axis map configurations when
 axes are shared between Motion Supervisors.  Since the Error conditions
 are automatically proliferated across mapped axes, it is possible to create
 Motion Supervisor axis maps that generate Error conditions that cannot be
 cleared.  For example, if MS#0 is mapped to axes 0 and 1, and MS#1 is mapped
 to axis 0.  MS#1 will NOT be able to clear Error conditions.  Only MS#0 will
 be able to clear Error conditions on axes 0 and 1.

Motion Supervisor axis maps are set with any of the following methods:

1)  mpiMotionAction(...) with the MEIActionMAP or MPIActionRESET
2)  mpiMotionStart(...)
3)  mpiMotionModify(...)
4)  mpiMotionEventNotifySet(...)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
```

```
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motMap1Main

argMainRENAME(main, motMap1)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)
#define MS_COUNT        (4)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber[MS_COUNT] = { 0, 1, 2, 3,  };
MPIMotionType   motionType  = MPIMotionTypeS_CURVE;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber[0],   },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 10000.0,  20000.0,    },
    { 0.0,      0.0,        },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent  */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
};

/* Motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};
```

```
MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

long
    motionDoneWait(MPIMotion motion);

long
    motionIdle(MPIMotion    motion,
               MPIStatus    *status);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* Motion controller handle */
    MPIAxis     axisX;      /* X axis */
    MPIAxis     axisY;      /* Y axis */

    MPIMotion   motion;     /* Motion object for no axes */
    MPIMotion   motionX;    /* Motion object for axisX */
    MPIMotion   motionY;    /* Motion object for axisY */
    MPIMotion   motionXY;   /* Motion object for axisX and axisY */

    enum {
        Motion,
        MotionX,
        MotionY,
        MotionXY,
    };

    long    index;

    long    returnValue;    /* Return value from library */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;
```

```
        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]   > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber[0] > (MEIXmpMAX_MSs - MS_COUNT)) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - MS_COUNT,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    motionNumber[1] = motionNumber[0] + 1;
    motionNumber[2] = motionNumber[1] + 1;
    motionNumber[3] = motionNumber[2] + 1;

    /* Create control object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
```

```
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create X axis object using AXIS_X number on controller */
    axisX =
        mpiAxisCreate(control,
                      axisNumber[0]);       /* Axis Number */
    msgCHECK(mpiAxisValidate(axisX));

    /* Create Y axis object using AXIS_Y number on controller */
    axisY =
        mpiAxisCreate(control,
                      axisNumber[1]);       /* Axis Number */
    msgCHECK(mpiAxisValidate(axisY));

    /* Create Motion object with NO axes */
    motion =
        mpiMotionCreate(control,
                        motionNumber[Motion],   /* Motion supervisor number */
                        NULL);
    msgCHECK(mpiMotionValidate(motion));

    /* Create Motion object for axisX */
    motionX =
        mpiMotionCreate(control,
                        motionNumber[MotionX],      /* Motion supervisor number */
                        axisX);
    msgCHECK(mpiMotionValidate(motionX));

    /* Create Motion object for axisY */
    motionY =
        mpiMotionCreate(control,
                        motionNumber[MotionY],      /* Motion supervisor number */
                        axisY);
    msgCHECK(mpiMotionValidate(motionY));

    /* Motion object for axisX and axisY */
    motionXY =
        mpiMotionCreate(control,
                        motionNumber[MotionXY],     /* Motion supervisor number */
                        axisX);
    msgCHECK(mpiMotionValidate(motionXY));

    /* Append axisY to motionXY */
    returnValue =
        mpiMotionAxisAppend(motionXY,
                            axisY);
    msgCHECK(returnValue);

    /* Map X axis to Motion Supervisor */
    returnValue =
        mpiMotionAction(motionX,        /* Axis X, using MotionX */
                        (MPIAction)MEIActionMAP);   /* Map axis to Motion Supervisor
 */
    msgCHECK(returnValue);

    /* Map Y axis to Motion Supervisor */
```

```
    returnValue =
        mpiMotionAction(motionY,        /* Axis Y, using MotionY */
                        (MPIAction)MEIActionMAP);   /* Map axis to Motion Supervisor
 */
    msgCHECK(returnValue);

    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIMotionParams motionParams;

        /* Clear any error conditions on axisX */
        returnValue =
            mpiMotionAction(motionX,
                            (MPIAction)MPIActionRESET);
        msgCHECK(returnValue);

        index = 0;

        /* Move axisX to position #1 */

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        printf("\nMoving X axis...\n");

        returnValue =
            mpiMotionStart(motionX,
                           motionType,
                           &motionParams);

        if (returnValue != MPIMessageOK) {
            printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                   motionX,
                   motionType,
                   &motionParams,
                   returnValue,
                   mpiMessage(returnValue, NULL));
        }

        /* Wait for motion to complete */
        returnValue = motionDoneWait(motionX);
        msgCHECK(returnValue);
```

```
        /* Clear any error conditions on axisY */
        returnValue =
            mpiMotionAction(motionY,
                            MPIActionRESET);
        msgCHECK(returnValue);

        /* Move axisY to position #1 */

        printf("\nMoving Y axis...\n");

        returnValue =
            mpiMotionStart(motionY,
                           motionType,
                           &motionParams);

        if (returnValue != MPIMessageOK) {
            printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                   motionY,
                   motionType,
                   &motionParams,
                   returnValue,
                   mpiMessage(returnValue, NULL));
        }

        /* Wait for motion to complete */
        returnValue = motionDoneWait(motionY);
        msgCHECK(returnValue);

        /* Configure Motion Supervisor Map for X,Y */
        returnValue =
            mpiMotionAction(motionXY,       /* Axes X and Y, using MotionXY */
                            (MPIAction)MEIActionMAP);   /* Map axes to Motion
 Supervisor */
        msgCHECK(returnValue);

        /* Clear any error conditions on AxisX and AxisY */
        returnValue =
            mpiMotionAction(motionXY,
                            MPIActionRESET);
        msgCHECK(returnValue);

        index = 1;

        /* Move axisX and axisY to position #2 */

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[index];
```

```
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        printf("\nMoving X and Y axes...\n");

        returnValue =
            mpiMotionStart(motionXY,
                           motionType,
                           &motionParams);

        if (returnValue != MPIMessageOK) {
            printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                   motionXY,
                   motionType,
                   &motionParams,
                   returnValue,
                   mpiMessage(returnValue, NULL));
        }

        /* Wait for motion to complete */
        returnValue = motionDoneWait(motionXY);
        msgCHECK(returnValue);

        /* Un-configure the controller's XY Motion Supervisor */
        returnValue =
            mpiMotionAction(motion,         /* Motion object has NO axes */
                            (MPIAction)MEIActionMAP);   /* Set a Motion Supervisor
map of no axes */
        msgCHECK(returnValue);
    }

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motionX);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motionY);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motionXY);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisY);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisX);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);
```

```
    return ((int)returnValue);
}

long
    motionDoneWait(MPIMotion motion)
{
    MPIStatus   status;

    long returnValue;

    long motionDone = FALSE;

    /* Poll status until motion done */
    while (motionDone == FALSE) {
        returnValue =
            mpiMotionStatus(motion,
                            &status,
                            NULL);
        msgCHECK(returnValue);

        switch (status.state) {
            case MPIStateIDLE: {
                motionDone = TRUE;

                returnValue =
                    motionIdle(motion,
                               &status);

                meiPlatformSleep(300);  /* msec */

                break;
            }
            case MPIStateERROR: {
                motionDone = TRUE;

                printf("Motion State Error!\n");

                break;
            }
            default: {
                break;
            }
        }
    }

    return (returnValue);
}

long
    motionIdle(MPIMotion    motion,
               MPIStatus    *status)
{
    long    returnValue;

    double          actual[AXIS_COUNT];
    double          command[AXIS_COUNT];
    MPITrajectory   trajectory[AXIS_COUNT];
```

```
    long    index;

    printf("MotionDone: status: state %d action %d eventMask 0x%x\n"
           "\tatTarget %d settled %d %s\n",
            status->state,
            status->action,
            status->eventMask,
            status->atTarget,
            status->settled,
            (status->settled == FALSE)
                    ? "=== NOT SETTLED ==="
                    : "");

    returnValue =
        mpiMotionPositionGet(motion,
                             actual,
                             command);
    msgCHECK(returnValue);

    returnValue =
        mpiMotionTrajectory(motion,
                            trajectory);
    msgCHECK(returnValue);


    /* Display axis positions */
    for (index = 0; index < AXIS_COUNT; index++) {
        printf("\taxis[%d]    position: command %11.3lf\tactual %11.3lf\n",
                index,
                command[index],
                actual[index]);
        printf("\t\ttrajectory: velocity %11.3lf\taccel  %11.3lf\tjerk %11.3lf\n",
                trajectory[index].velocity,
                trajectory[index].acceleration,
                trajectory[index].jerkPercent);
    }

    return (returnValue);
}
```

motmap2.c -- Motion object and Motion Supervisor Axis mapping with events

```
/* motMap2.c  */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Motion object and Motion Supervisor Axis mapping with events

This sample code performs the following operation:
 -Map axis 0 to motion object 'motion'.
 -Move axis 0 to position.
 -Wait for an event based upon motion done on axis 0
 -Move axis 1 to position
 -Wait for an event based upon motion done on axis 1
 -Map axis 0 and axis 1 to 'motion'
 -Coordinated move on axis 0 and axis 1 back to position (0,0).

Only one Motion Supervisor and Motion object is used and remapped after
 each move.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motMap2Main

argMainRENAME(main, motMap2)
#endif
```

```
#define AXIS_COUNT      (2)
#define MS_COUNT        (1)
#define MOTION_COUNT    (3)
#define COUNT           (100)


/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionSupNumber[MS_COUNT] = { 0,};

MPIMotionType   motionType  = MPIMotionTypeS_CURVE;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionSupNumber[0], },
    {   "-type",    ArgTypeLONG,    &motionType, },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 10000.0,},
    { 10000.0,},
    { 0.0,},
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent  */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
};

/* Motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
    {   &trajectory[2][0],  &position[2][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
```

```
    {   &trajectory[2][0],  &position[2][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
    {   &trajectory[2][0],  },
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;                /* Motion controller handle */
    MPIAxis         axisList[AXIS_COUNT];   /* Axis handle(s) */
    MPIMotion       motion;                 /* Motion object */

    MPIMotionParams motionParams;

    MPINotify       notify;                 /* event notification object */
    MPIEventMgr     eventMgr;               /* event manager handle */

    MPIEventMask    eventMask;
    MPIEventStatus  eventStatus;

    long        returnValue;    /* Return value from library */
    long        argIndex;
    Service     service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
```

```
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]   > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionSupNumber[0] > (MEIXmpMAX_MSs - MS_COUNT)) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - MS_COUNT,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /* Create control object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    axisNumber[1] = axisNumber[0] + 1;

    /* Create axis object using axisNumber on controller */
    axisList[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);
```

```
    msgCHECK(mpiAxisValidate(axisList[0]));

    /* Create axis object using axisNumber on controller  */
    axisList[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axisList[1]));

    /* Create motion object motion for axisList[0] */
    motion =
        mpiMotionCreate(control,
                        0,                  /* Motion supervisor number */
                        axisList[0]);
    msgCHECK(mpiMotionValidate(motion));

    /* Create event notification object for events from all sources */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);
    notify =
        mpiNotifyCreate(eventMask,
                        NULL);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr =
        mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Flush any existing events */
    returnValue =
        mpiEventMgrFlush(eventMgr);
    msgCHECK(returnValue);

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Request notification of events from motion */
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
```

```
    meiASSERT(service != NULL);


    /* Clear any error conditions on motion */
    returnValue =
        mpiMotionAction(motion,
                        MPIActionRESET);
    msgCHECK(returnValue);

    /* Move axis 0 */
    switch (motionType) {
        case MPIMotionTypeS_CURVE: {
            motionParams.sCurve = sCurve[0];
            break;
        }
        case MPIMotionTypeTRAPEZOIDAL: {
            motionParams.trapezoidal = trapezoidal[0];
            break;
        }
        case MPIMotionTypeVELOCITY: {
            motionParams.velocity = velocity[0];
            break;
        }
        default: {
            meiASSERT(FALSE);
            break;
        }
    }

    printf("\nMoving axis 0...\n");

    returnValue =
        mpiMotionStart(motion,
                       motionType,
                       &motionParams);

    if (returnValue != MPIMessageOK) {
        printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
               motion,
               motionType,
               &motionParams,
               returnValue,
               mpiMessage(returnValue, NULL));
    }

    /* Wait for motion to complete */
    while (returnValue == MPIMessageOK) {

        /* Wait for axis and motion event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
```

```
                                MPIWaitFOREVER);

        fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            printf("Finished axis 0 move.\n");
            break;
        }
        else {
            continue;
        }
    }

    /* Map axisList[1] to motion object */
    returnValue =
        mpiMotionAxisListSet(motion,
                             1,
                             &axisList[1]);
    msgCHECK(returnValue);

    /* Request notification of events from motion */
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Move axis 1 */
    switch (motionType) {
        case MPIMotionTypeS_CURVE: {
            motionParams.sCurve = sCurve[1];
            break;
        }
        case MPIMotionTypeTRAPEZOIDAL: {
            motionParams.trapezoidal = trapezoidal[1];
            break;
        }
        case MPIMotionTypeVELOCITY: {
            motionParams.velocity = velocity[1];
            break;
        }
        default: {
```

```
            meiASSERT(FALSE);
            break;
        }
    }

    printf("\nMoving Axis 1...\n");

    returnValue =
        mpiMotionStart(motion,
                       motionType,
                       &motionParams);

    if (returnValue != MPIMessageOK) {
        printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
               motion,
               motionType,
               &motionParams,
               returnValue,
               mpiMessage(returnValue, NULL));
    }

    /* Wait for motion to complete */
    while (returnValue == MPIMessageOK) {

        /* Wait for axis and motion event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);

        fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            printf("Finished axis 1 move.\n");
            break;
        }
        else {
            continue;
        }
    }

    /* Map motion for axis 0 and axis 1 */
    returnValue =
```

```
        mpiMotionAxisListSet(motion,
                             2,
                             axisList);
    msgCHECK(returnValue);

    /* Request notification of events from motion */
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    switch (motionType) {
        case MPIMotionTypeS_CURVE: {
            motionParams.sCurve = sCurve[2];
            break;
        }
        case MPIMotionTypeTRAPEZOIDAL: {
            motionParams.trapezoidal = trapezoidal[2];
            break;
        }
        case MPIMotionTypeVELOCITY: {
            motionParams.velocity = velocity[2];
            break;
        }
        default: {
            meiASSERT(FALSE);
            break;
        }
    }

    printf("\nMoving Axes 0 and 1...\n");

    returnValue =
        mpiMotionStart(motion,
                       motionType,
                       &motionParams);

    if (returnValue != MPIMessageOK) {
        printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
               motion,
               motionType,
               &motionParams,
               returnValue,
               mpiMessage(returnValue, NULL));
    }

    /* Wait for motion to complete */
    while (returnValue == MPIMessageOK) {

        /* Wait for axis and motion event */
        returnValue =
```

```
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);

        fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            printf("Finished axis 0 and 1 move.\n");
            break;
        }
        else {
            continue;
        }
    }

    /* Delete Objects */
    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisList[0]);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisList[1]);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

motmod1.c -- Point to Point trapezoidal profile motion with end point modification.

```
/* motMod1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/motmod1.c 4     7/23/01 2:36p Kevinh $";
#endif

/*

:Point to Point trapezoidal profile motion with end point modification.

This is a simple program to demonstrate how to start a trapezoidal profile
 motion on a single axis.  During the execution of the motion, the target
 position (end point), velocity, accel, decel are modified.  The controller
 automatically re-calculates the motion profile, using the modified
 trajectory parameters.

The mpiMotionModify(...) function modifies a presently executing motion
 profile.  If mpiMotionModify(...) is called when no motion is executing,
 the error code, MPIMotionMessageIDLE will be returned, and no motion
 will be commanded.  This error code let's the application know that there
 was no motion to be modified.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motMod1Main

argMainRENAME(main, motMod1)
#endif
```

```
#define MOTION_NUMBER       (0)
#define AXIS_NUMBER         (0)

/* Motion Start Parameters */
#define TARGET_POSITION     (10000.0)
#define VEL                 (10000.0)
#define ACCEL               (100000.0)
#define DECEL               (100000.0)

/* Motion Modify Parameters */
#define MODIFY_POSITION     (-1000.0)   /* new target position */
#define MODIFY_VEL          (5000.0)    /* new velocity */
#define MODIFY_DECEL        (50000.0)   /* new deceleration */

#define WAIT_TIME           (500)       /* delay to modify move, (msec) */


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller object handle */
    MPIMotion   motion;     /* motion object handle */
    MPIAxis     axis;       /* axis object handle */

    long    returnValue;

    MPIMotionParams params;     /* motion parameters */
    MPITrajectory   trajectory; /* motion trajectory */
    double          position;   /* final target position */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n",
                           argv[0],
                           ArgUSAGE);
        exit(MPIMessagePARAM_INVALID);
    }


    /* Create motion controller object*/
    control =
        mpiControlCreate(MPIControlTypeDEFAULT,
                         NULL);
```

```
    msgCHECK(mpiControlValidate(control));

    /* Initialize the motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object*/
    axis =
        mpiAxisCreate(control,
                      AXIS_NUMBER);     /* axis number */
    msgCHECK(mpiAxisValidate(axis));

    /* Create motion object, append axis */
    motion =
        mpiMotionCreate(control,
                        MOTION_NUMBER,  /* motion supervisor number */
                        axis);          /* axis object handle */
    msgCHECK(mpiMotionValidate(motion));

    /* Set up motion parameters */
    trajectory.velocity     = VEL;          /* counts per sec */
    trajectory.acceleration = ACCEL;        /* counts per sec * sec */
    trajectory.deceleration = DECEL;

    position = TARGET_POSITION;     /* target position (counts) */

    params.trapezoidal.trajectory   = &trajectory;
    params.trapezoidal.position     = &position;

    printf("\nMotionStart...\n");

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
                       &params);
    msgCHECK(returnValue);

    meiPlatformSleep(WAIT_TIME);    /* delay */

    /* Set up new motion parameters */
    trajectory.velocity     = MODIFY_VEL;       /* counts per sec */
    trajectory.acceleration = ACCEL;            /* use same acceleration */
    trajectory.deceleration = MODIFY_DECEL;     /* counts per sec * sec */

    /* New target position (counts) */
    position = MODIFY_POSITION;

    printf("\nMotionModify...\n");

    /* Modify the executing motion profile */
    returnValue =
        mpiMotionModify(motion,
                        MPIMotionTypeTRAPEZOIDAL,
```

```
                        &params);
    msgCHECK(returnValue);

    /* object clean-up */
    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

motorio1.c -- Configure Transceiver as input or output and toggle.

```
/* motorIo1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/motorio1.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Configure Transceiver as input or output and toggle.

This sample application configures a single transceiver, on one motor,
for input or output.  If the transceiver is configured for output, the
bit state is configured by changing the user defined XCVR_STATE value.
As a tutorial purpose TRANSCEIVER_ID and TRANSCEIVER_MASK are defined
using MPI definitions.  The MPI definitions must have the same ending
transceiver letter!!(i.e. XCVR_A, XCVR_B, or XCVR_C)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    motorIo1Main

argMainRENAME(main, motorIo1)
#endif

/* Command line arguments and defaults */
long    motorNumber = 0;

Arg argList[] = {
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};
```

```
/* User Settings */
#define IO_CONFIG           (MEIMotorTransceiverConfigOUTPUT) /* INPUT or OUTPUT  */
#define TRANSCEIVER_ID      (MEIMotorTransceiverIdA)  /* A, B, or C               */
#define TRANSCEIVER_MASK    (MEIMotorTransceiverMaskA) /* same as above (A,B,or C) */

#define INVERT_BIT          (FALSE)
#define XCVR_STATE          (TRUE)  /* State of output transceiver */

#define WAIT_TIME           (10)    /* Units are msec */
int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;

    MPIMotor        motor;
    MEIMotorConfig  motorConfigXmp;  /* Contains transceiver configuration */
    MPIMotorIo      io;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    returnValue;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-motor # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
```

```
    }

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Get handle to motor object */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    returnValue = mpiMotorValidate(motor);
    msgCHECK(returnValue);

    /* Configure selected transceiver */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Transceiver[TRANSCEIVER_ID].Config = IO_CONFIG;
    motorConfigXmp.Transceiver[TRANSCEIVER_ID].Invert = INVERT_BIT;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /*
       Write the output word if the selected transceiver is configured
       for output.
    */
    if (IO_CONFIG == MEIMotorTransceiverConfigOUTPUT) {
        returnValue =
            mpiMotorIoGet(motor,
                          &io);
        msgCHECK(returnValue);

        if (io.output & TRANSCEIVER_MASK) {
            io.output = (io.output & ~TRANSCEIVER_MASK);
        } else {
            io.output = (io.output | TRANSCEIVER_MASK);
        }

        returnValue =
            mpiMotorIoSet(motor,
                          &io);
        msgCHECK(returnValue);
    }

    /* Wait 10ms before reading new configuration */
```

```
    meiPlatformSleep(WAIT_TIME);

    /* Read and display the current transceiver IO bit */
    returnValue =
        mpiMotorIoGet(motor,
                      &io);
    msgCHECK(returnValue);

    printf("\n%s: %d\n",
           (IO_CONFIG == MEIMotorTransceiverConfigINPUT)
                ? "Input bit"
                : "Output bit",
           (IO_CONFIG == MEIMotorTransceiverConfigINPUT)
                ? (io.input & TRANSCEIVER_MASK)
                : (io.output & TRANSCEIVER_MASK));


    /* Delete the MOTOR handle */
    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    /* Delete the CONTROL handle */
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

notify1.c -- Create notify object and wait for events.

```
/* notify1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/notify1.c 20    7/23/01 2:36p Kevinh $";
#endif

/*

:Create notify object and wait for events.

Create a thread that uses a notify object to wait for user-defined events
 passed to the notify object from the main thread.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "stdmpi.h"
#include "stdmei.h"
#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    notify1Main

argMainRENAME(main, notify1)
#endif

/* Command line arguments and defaults */

Arg argList[] = {
    {   NULL,       ArgTypeINVALID, NULL,   }
};

Static  long    NotifyWaitAwake;

Static long
    notifyWait(MPINotify notify);
```

```
int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;        /* Motion controller handle */
    MPINotify   notify;         /* Event notification object */

    MPIEventMask    eventMask;

    long    returnValue;    /* Return value from library */

    long    eventDone;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    Thread          thread;
    ThreadStatus    status;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n",
                           argv[0],
                           ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));
```

```
    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Request notification of ALL events */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);

    /* Create event notification object */
    notify =
        mpiNotifyCreate(eventMask,
                        NULL);
    msgCHECK(mpiNotifyValidate(notify));

    if (returnValue == MPIMessageOK) {
        thread = threadCreate();

        returnValue = threadValidate(thread);
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            threadStart(thread,
                        (THREAD_FUNCTION)notifyWait,
                        notify);
    }

    eventDone = FALSE;

    while ((returnValue == MPIMessageOK) &&
           (eventDone == FALSE)) {
        char    buffer[128];
        long    index;
        long    haveEventNumber;

        /* Wait for notifyWait() to call mpiNotifyEventWait() */
        while (NotifyWaitAwake != FALSE) {
            meiPlatformSleep(1);
        }

        /* Make sure notifyWait() thread is still executing */
        returnValue =
            threadStatus(thread,
                         &status);

        if (returnValue != MPIMessageOK) {
            break;
        }

        if (status.active == FALSE) {
            break;
        }

        meiPlatformConsole("Enter an event number, or ESC to exit ...\n");

        memset(buffer, 0, sizeof(buffer));
        index = 0;
```

```
        haveEventNumber = FALSE;
        while (haveEventNumber == FALSE) {
            MPIEventStatus  eventStatus;

            long    key;
            long    eventNumber = 0;

            key = meiPlatformKey(MPIWaitFOREVER);


            if (key <= 0) {
                meiPlatformKey(MPIWaitFOREVER);
                continue;
            }

            switch (key) {
                case 0x1b: {    /* ESC */
                    eventNumber = key;
                    haveEventNumber = TRUE;
                    eventDone = TRUE;
                    break;
                }
                case '\r':
                case '\n': {
                    meiPlatformConsole("\n");

                    eventNumber = meiPlatformAtol(buffer);

                    haveEventNumber = TRUE;
                    break;
                }
                case '\b': {
                    if (index > 0) {
                        meiPlatformConsole("\b \b");
                        buffer[--index] = '\0';
                    }
                    break;
                }
                case '-': {
                    if (index == 0) {
                        buffer[index++] = (char)key;
                        meiPlatformConsole("%c", key);
                    }
                    break;
                }
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9': {
                    buffer[index++] = (char)key;
```

```
                    meiPlatformConsole("%c", key);
                    break;
                }
                default: {
                    break;
                }
            }

            if (haveEventNumber != FALSE) {
                NotifyWaitAwake = TRUE;

                eventStatus.type    = MPIEventTypeEXTERNAL;
                eventStatus.source  = NULL;
                eventStatus.info[0] = eventNumber;

                returnValue =
                    mpiNotifyEventWake(notify,
                                       &eventStatus);
            }
        }
    }

    if (thread != NULL) {
        while (returnValue == MPIMessageOK) {
            returnValue =
                threadStatus(thread,
                             &status);
            msgCHECK(returnValue);

            if (status.active == FALSE) {
                returnValue = status.returnValue;
                break;
            }

            meiPlatformSleep(1);
        }

        returnValue = threadDelete(thread);
        msgCHECK(returnValue);
    }

    fprintf(stderr,
            "%s exiting: returnValue 0x%x: %s\n",
            argv[0],
            returnValue,
            mpiMessage(returnValue, NULL));

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

Static long
```

```
    notifyWait(MPINotify notify)
{
    long    returnValue;

    MPIWait timeout;
    long    eventDone;

    /* Validate the notify object and its handle */
    returnValue = mpiNotifyValidate(notify);

    timeout = MPIWaitFOREVER;

    eventDone = FALSE;

    while ((returnValue == MPIMessageOK) &&
           (eventDone == FALSE)) {
        MPIEventStatus  eventStatus;

        NotifyWaitAwake = FALSE;

        /* Wait for notify event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               timeout);

        meiPlatformConsole("mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x: %s\n",
                           notify,
                           &eventStatus,
                           timeout,
                           returnValue,
                           mpiMessage(returnValue, NULL));

        switch (returnValue) {
            case MPIMessageOK: {
                meiPlatformConsole("\teventStatus: type %d number %d\n",
                                   eventStatus.type,
                                   eventStatus.info[0]);

                if (eventStatus.info[0] == 0x1b) {
                    eventDone = TRUE;
                }
                break;
            }
            case MPIMessageTIMEOUT: {
                meiPlatformConsole("\tTIMEOUT\n");
                break;
            }
            default: {
                break;
            }
        }
    }

    meiPlatformConsole("notifyWait(0x%x) returning 0x%x: %s\n",
                       notify,
                       returnValue,
```

```
                        mpiMessage(returnValue, NULL));

    return (returnValue);
}
```

path1.c -- Two-axis path motion, around a rectangle with rounded corners.

```
/* path1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Two-axis path motion, around a rectangle with rounded corners.

This sample demonstrates how to create a two-axis path motion.

After the Path object has been created, Line and Arc elements are
 appended to the path.  After the path elements have been appended,
 the MotionParams are generated, and the motion is started using
 mpiMotionStart(...).

When the motion has started, the program will wait for a MOTION_DONE event
 to be distributed by the event manager in a separate service thread.

A summary of Path object structures and methods:

    MPIPath -- path object handle
    MPIPathConfig{} -- path config structure
    MPIPathArc{} -- arc from start, angle and radius
    MPIPathLine{} -- point in space

    mpiPathCreate() -- create a path
    mpiPathDelete() -- delete a path
    mpiPathValidate() -- validate a path
    mpiPathConfigGet() -- get the current path configuration
    mpiPathConfigSet() -- set the path configuration
    mpiPathAppend() -- add a path element to the path
    mpiPathMotionParamsGenerate() -- generate the motion parameters


Note:  When multiple axes are associated with a motion supervisor, the
 controller automatically combines the individual axis and motor status
 into the motion status.  Thus, if a Stop, E-Stop or Abort action occurs
 on one axis, the event will be propogated automatically to the other axes.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
```

```
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#define AXIS_COUNT          (2)

/* Set the vector values for Vel, Accel, Decel */
#define VELOCITY            (5000.0)
#define ACCEL               (50000.0)
#define DECEL               (50000.0)

/* The practical range for the time slice parameter is from
    10 msec (0.01) to 100 msec (0.1).  The time slice determines
    the spacing of points for the internal algorithm interpolation */
#define TIME_SLICE          (0.050) /* seconds */

/* There must be this many points in the buffer, or the motion
    will E_STOP */
#define EMPTY_COUNT         (5)

/* Ensure that the E_STOP will finish before the points are empty */
#define E_STOP_RATE         (TIME_SLICE * EMPTY_COUNT)

/* Only BSPLINE and BLSPLINE2 are supported for Path motion */
#define MOTION_TYPE         MPIMotionTypeBSPLINE

#define TIMEOUT             (10000) /* ms to wait for MOTION_DONE */


#if defined(ARG_MAIN_RENAME)
#define main    path1Main

argMainRENAME(main, path1)
#endif

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   NULL,       ArgTypeINVALID, NULL,   }
```

```
};


void main(int argc, char *argv[])
{
    MPIControl          control;        /* motion controller handle */
    MPIAxis             axis[AXIS_COUNT];       /* axis handle(s) */
    MPIMotion           motion;         /* motion handle */
    MPINotify           notify;         /* event notification handle */
    MPIEventMgr         eventMgr;       /* event manager handle */

    MPIMotionConfig     motionConfig;
    MPIPath             path;
    MPIPathConfig       pathConfig;
    MPIPathElement      element;
    MPIMotionParams     motionParams;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIEventMask        eventMask;

    Service             service;        /* service handle */

    long                returnValue;    /* return value from library */
    long                argIndex;
    double              x_start;
    double              y_start;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
```

```
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis objects */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis[0]));

    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axis[1]));

    /* Create motion supervisor object using MS number 0 */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create 2-axis motion coordinate system */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axis);
    msgCHECK(returnValue);

    /* Request notification of all MPI events from motion */
    mpiEventMaskCLEAR(eventMask);
```

```
    mpiEventMaskALL(eventMask);

    /* Request notification of all events from motion */
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* default sleep (msec) */
    meiASSERT(service != NULL);

    /* Read current motion supervisor configuration */
    returnValue =
        mpiMotionConfigGet(motion,
                           &motionConfig,
                           NULL);
    msgCHECK(returnValue);

    /* Set new E_STOP deceleration rates */
    motionConfig.decelTime.eStop = E_STOP_RATE;

    returnValue =
        mpiMotionConfigSet(motion,
                           &motionConfig,
                           NULL);
    msgCHECK(returnValue);

    returnValue = mpiAxisCommandPositionGet(axis[0], &x_start);
    msgCHECK(returnValue);
    returnValue = mpiAxisCommandPositionGet(axis[1], &y_start);
    msgCHECK(returnValue);
```

```
    path = mpiPathCreate();
    msgCHECK(mpiPathValidate(path));

    returnValue = mpiPathConfigGet(path,
                                   &pathConfig,
                                   NULL);
    msgCHECK(returnValue);

    pathConfig.dimension = AXIS_COUNT;
    MPIPathPointX(pathConfig.start) = x_start;
    MPIPathPointY(pathConfig.start) = y_start;
    pathConfig.velocity = VELOCITY;
    pathConfig.acceleration = ACCEL;
    pathConfig.deceleration = DECEL;
    pathConfig.interpolation = MOTION_TYPE;
    pathConfig.timeSlice = TIME_SLICE;

    returnValue = mpiPathConfigSet(path,
                                   &pathConfig,
                                   NULL);
    msgCHECK(returnValue);


    /* Start appending elements to the path */

    element.type = MPIPathElementTypeLINE;

    /* Set blending to TRUE if the corners of the path
        should be rounded.   Set to FALSE if the corners
        should be sharp. The motion will stop at corners when
        blending is FALSE */
    element.blending = TRUE;
    MPIPathPointX(element.params.line.point) = 1800.0;
    MPIPathPointY(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC;
    element.params.arc.angle.start = 270.0;
    /* Included angle is CCW when positive, CW when negative */
    element.params.arc.angle.included = 90.0;
    element.params.arc.radius = 200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = 2000.0;
```

```
    MPIPathPointY(element.params.line.point) = 1800.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC;
    element.params.arc.angle.start = 0.0;
    element.params.arc.angle.included = 90.0;
    element.params.arc.radius = 200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = -1800.0;
    MPIPathPointY(element.params.line.point) = 2000.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC;
    element.params.arc.angle.start = 90.0;
    element.params.arc.angle.included = 90.0;
    element.params.arc.radius = 200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = -2000.0;
    MPIPathPointY(element.params.line.point) = 200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC;
    element.params.arc.angle.start = 180.0;
    element.params.arc.angle.included = 90.0;
    element.params.arc.radius = 200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = 0.0;
```

```
    MPIPathPointY(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);
    /* Finished appending elements to the path */

    /* Generate motion parameters from the path object */
    returnValue = mpiPathMotionParamsGenerate(path,
                                      &motionParams);
    msgCHECK(returnValue);

    /* Keep the points in memory, in case we need to backup on path */
    motionParams.bspline.point.retain = TRUE;

    /* Specify the minimum number of points required in
        the XMP buffer -- if there are less, motion will E_STOP */
    motionParams.bspline.point.emptyCount = EMPTY_COUNT;

    /* This motion will not be appended */
    motionParams.bspline.point.final = TRUE;

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MOTION_TYPE,
                       &motionParams);
    fprintf(stderr,
            "mpiMotionStart returns 0x%x: %s\n",
            returnValue,
            mpiMessage(returnValue, NULL));

    /* Collect motion events */
    while (returnValue == MPIMessageOK) {
        MPIEventStatus  eventStatus;

        /* Wait for event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               TIMEOUT);
        msgCHECK(returnValue);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            printf("Motion Done\n");
            break;
        }
        else {
            fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
```

```
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        }
    }

    fprintf(stderr,
            "%s exiting: returnValue 0x%x: %s\n",
            argv[0],
            returnValue,
            mpiMessage(returnValue, NULL));

    returnValue = mpiPathDelete(path);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis[0]);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis[1]);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);
}
```

path3d1.c -- Three-axis path motion with velocities, acceleration and timeSlices

```
/* path1_3D.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Three-axis path motion with velocities, acceleration and timeSlices
defined for each element.

This sample demonstrates how to create a three-axis path motion together wth
different velocities, aceleration and timeSlices for different elements.

The code draws 2 rectangles. The first, larger rectangle has different velocities
and timeSlices, and different z elevations. The second, smaller one is simply
in the xy plane (z = 0) and has one corner with the direction of the arc
value = -1 which means that the motion will be around 270 degrees and not 90.
Even though the second rectangle is 2D, it must have values for z as it is
inadvisable
to mix 2D and 3D paths.

After the Path object has been created, Line and Arc elements are
 appended to the path.  After the path elements have been appended,
 the MotionParams are generated, and the motion is started using
 mpiMotionStart(...).

When the motion has started, the program will wait for a MOTION_DONE event
 to be distributed by the event manager in a separate service thread.

A summary of Path object structures and methods:

    MPIPath -- path object handle
    MPIPathParams{} -- path Params structure
    MPIPathArc{} -- arc from start, angle and radius
    MPIPathLine{} -- point in space

    mpiPathCreate() -- create a path
    mpiPathDelete() -- delete a path
    mpiPathValidate() -- validate a path
    mpiPathParamsGet() -- get the current path parameters
    mpiPathParamsSet() -- set the path parameters
    mpiPathAppend() -- add a path element to the path
    mpiPathMotionParamsGenerate() -- generate the motion parameters


Note:  When multiple axes are associated with a motion supervisor, the
 controller automatically combines the individual axis and motor status
 into the motion status.  Thus, if a Stop, E-Stop or Abort action occurs
```

```
 on one axis, the event will be propogated automatically to the other axes.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#define AXIS_COUNT          (3)

/* Set the vector values for Vel, ACCEL, Decel */
#define VELOCITY            (5000.0)
#define ACCEL               (50000.0)
#define DECEL               (50000.0)

/* The practical range for the time slice parameter is from
    10 msec (0.01) to 100 msec (0.1).  The time slice determines
    the spacing of points for the internal algorithm interpolation */
#define TIME_SLICE          (0.050) /* seconds */

/* There must be this many points in the buffer, or the motion
    will E_STOP */
#define EMPTY_COUNT         (5)

/* Ensure that the E_STOP will finish before the points are empty */
#define E_STOP_RATE         (TIME_SLICE * EMPTY_COUNT)

/* Only BSPLINE and BLSPLINE2 are supported for Path motion */
#define MOTION_TYPE         MPIMotionTypeBSPLINE

#define TIMEOUT             (100000) /* ms to wait for MOTION_DONE */


#if defined(ARG_MAIN_RENAME)
#define main    path1Main

argMainRENAME(main, path1)
#endif

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, 2};
long            motionNumber    = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   NULL,       ArgTypeINVALID, NULL,   }
};
```

```
void main(int argc, char *argv[])
{
    MPIControl          control;        /* motion controller handle */
    MPIAxis             axis[AXIS_COUNT];       /* axis handle(s) */
    MPIMotion           motion;         /* motion handle */
    MPINotify           notify;         /* event notification handle */
    MPIEventMgr         eventMgr;       /* event manager handle */

    MPIMotionConfig     motionConfig;
    MPIPath             path;
    MPIPathParams       pathParams;
    MPIPathElement      element;
    MPIMotionParams     motionParams;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIEventMask        eventMask;

    Service             service;        /* service handle */

    long                returnValue;    /* return value from library */
    long                argIndex;
    double              x_start;
    double              y_start;
    double              z_start;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
```

```
                        argv[0],
                        ArgUSAGE,
                        MEIXmpMAX_Axes - AXIS_COUNT,
                        MEIXmpMAX_MSs - 1);
    exit(MPIMessageARG_INVALID);
}

axisNumber[1] = axisNumber[0] + 1;
axisNumber[2] = axisNumber[1] + 1;

/* Create motion controller object */
control =
    mpiControlCreate(controlType,
                     &controlAddress);
msgCHECK(mpiControlValidate(control));

/* Initialize the controller */
returnValue = mpiControlInit(control);
msgCHECK(returnValue);

/* Create axis objects */
axis[0] =
    mpiAxisCreate(control,
                  axisNumber[0]);
msgCHECK(mpiAxisValidate(axis[0]));

axis[1] =
    mpiAxisCreate(control,
                  axisNumber[1]);
msgCHECK(mpiAxisValidate(axis[1]));

axis[2] =
    mpiAxisCreate(control,
                  axisNumber[2]);
msgCHECK(mpiAxisValidate(axis[2]));

/* Create motion supervisor object using MS number 0 */
motion =
    mpiMotionCreate(control,
                    motionNumber,
                    MPIHandleVOID);
msgCHECK(mpiMotionValidate(motion));

/* Create 2-axis motion coordinate system */
returnValue =
    mpiMotionAxisListSet(motion,
                         AXIS_COUNT,
                         axis);
msgCHECK(returnValue);

/* Request notification of all MPI events from motion */
mpiEventMaskCLEAR(eventMask);
mpiEventMaskALL(eventMask);

/* Request notification of all events from motion */
returnValue =
    mpiMotionEventNotifySet(motion,
```

```
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* default sleep (msec) */
    meiASSERT(service != NULL);

    /* Read current motion supervisor Configurationuration */
    returnValue =
        mpiMotionConfigGet(motion,
                           &motionConfig,
                           NULL);
    msgCHECK(returnValue);

    /* Set new E_STOP deceleration rates */
    motionConfig.decelTime.eStop = E_STOP_RATE;

    returnValue =
        mpiMotionConfigSet(motion,
                           &motionConfig,
                           NULL);
    msgCHECK(returnValue);

    returnValue = mpiAxisCommandPositionGet(axis[0], &x_start);
    msgCHECK(returnValue);
    returnValue = mpiAxisCommandPositionGet(axis[1], &y_start);
    msgCHECK(returnValue);
    returnValue = mpiAxisCommandPositionGet(axis[2], &z_start);
    msgCHECK(returnValue);

    path = mpiPathCreate();
    msgCHECK(mpiPathValidate(path));

    returnValue = mpiPathParamsGet(path,
                                   &pathParams,
                                   NULL);
    msgCHECK(returnValue);
```

```
    pathParams.dimension = AXIS_COUNT;
    MPIPathPointX(pathParams.start) = x_start;
    MPIPathPointY(pathParams.start) = y_start;
    MPIPathPointZ(pathParams.start) = z_start;
    pathParams.velocity = VELOCITY;
    pathParams.acceleration = ACCEL;
    pathParams.deceleration = DECEL;
    pathParams.interpolation = MOTION_TYPE;
    pathParams.timeSlice = TIME_SLICE;

    returnValue = mpiPathParamsSet(path,
                                   &pathParams,
                                   NULL);
    msgCHECK(returnValue);


    /* Start appending elements to the path */

    /* In this element, the path parameters will be used */
    element.type = MPIPathElementTypeLINE ;

    /*
        Set blending to TRUE if the corners of the path
        should be rounded.   Set to FALSE if the corners
        should be sharp. The motion will stop at corners when
        blending is FALSE
    */
    element.blending = TRUE;
    MPIPathPointX(element.params.line.point) = 1800.0;
    MPIPathPointY(element.params.line.point) = 0.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    /*
        The "| MPIPathElementAttrMaskVELOCITY" in the next line states that
        the element has it's own velocity.
    */
    element.type = MPIPathElementTypeARC_END_POINT | MPIPathElementAttrMaskVELOCITY;

    MPIPathPointX(element.params.arcEndPoint.center) = 1800;
    MPIPathPointY(element.params.arcEndPoint.center) = 200;
    MPIPathPointZ(element.params.arcEndPoint.center) = 50;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = 2000;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = 200;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 100;

    element.attributes.velocity = 10 * VELOCITY;

    /*
        The 'direction' is used to describe whether the motion should be around the
large,
        or small angle between the start and stop points. direction = 1 means that
```

```
        the motion will be through less than 180 degrees, while direction = -1 means
 that
        the motion will be through more than 180 degrees.
    */
    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    /*
        The "| MPIPathElementAttrMaskTIMESLICE" in the next line states that
        the element has it's own velocity.
    */
    element.type = MPIPathElementTypeLINE | MPIPathElementAttrMaskTIMESLICE;
    MPIPathPointX(element.params.line.point) = 2000.0;
    MPIPathPointY(element.params.line.point) = 1800.0;
    MPIPathPointZ(element.params.line.point) = 100.0;

    element.attributes.timeSlice = 0.1 * TIME_SLICE;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    /* In this element, the path parameters will be used */
    element.type = MPIPathElementTypeARC_END_POINT;
    MPIPathPointX(element.params.arcEndPoint.center) = 1800;
    MPIPathPointY(element.params.arcEndPoint.center) = 1800;
    MPIPathPointZ(element.params.arcEndPoint.center) = 50;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = 1800;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = 2000;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 0;


    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE ;
    MPIPathPointX(element.params.line.point) = -1800.0;
    MPIPathPointY(element.params.line.point) = 2000.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    element.type = MPIPathElementTypeARC_END_POINT;
    MPIPathPointX(element.params.arcEndPoint.center) = -1800;
    MPIPathPointY(element.params.arcEndPoint.center) = 1800;
    MPIPathPointZ(element.params.arcEndPoint.center) = -100;
```

```
    MPIPathPointX(element.params.arcEndPoint.endPoint) = -2000;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = 1800;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = -200;

    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = -2000.0;
    MPIPathPointY(element.params.line.point) = 200.0;
    MPIPathPointZ(element.params.line.point) = -200.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    element.type = MPIPathElementTypeARC_END_POINT;
    MPIPathPointX(element.params.arcEndPoint.center) = -1800;
    MPIPathPointY(element.params.arcEndPoint.center) = 200;
    MPIPathPointZ(element.params.arcEndPoint.center) = -100;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = -1800;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = 0;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 0;

    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = 0.0;
    MPIPathPointY(element.params.line.point) = 0.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    element.type = MPIPathElementTypeLINE ;
    MPIPathPointX(element.params.line.point) = 0.0;
    MPIPathPointY(element.params.line.point) = -1800.0;
    MPIPathPointZ(element.params.line.point) = 0.0;


    returnValue = mpiPathAppend(path,
                                &element);

    element.type = MPIPathElementTypeARC_END_POINT ;
    MPIPathPointX(element.params.arcEndPoint.center) = -200;
    MPIPathPointY(element.params.arcEndPoint.center) = -1800;
```

```
    MPIPathPointZ(element.params.arcEndPoint.center) = 0;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = -200;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = -2000;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 0;

    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    msgCHECK(returnValue);
        element.type = MPIPathElementTypeLINE ;
    MPIPathPointX(element.params.line.point) = -1800.0;
    MPIPathPointY(element.params.line.point) = -2000.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC_END_POINT ;
    MPIPathPointX(element.params.arcEndPoint.center) = -1800;
    MPIPathPointY(element.params.arcEndPoint.center) = -1800;
    MPIPathPointZ(element.params.arcEndPoint.center) = 0;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = -2000;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = -1800;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 0;

    element.params.arcEndPoint.direction = -1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    element.type = MPIPathElementTypeLINE ;
    MPIPathPointX(element.params.line.point) = -2000.0;
    MPIPathPointY(element.params.line.point) = -200.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);

    element.type = MPIPathElementTypeARC_END_POINT;
    MPIPathPointX(element.params.arcEndPoint.center) = -1800;
    MPIPathPointY(element.params.arcEndPoint.center) = -200;
    MPIPathPointZ(element.params.arcEndPoint.center) = 0;

    MPIPathPointX(element.params.arcEndPoint.endPoint) = -1800;
    MPIPathPointY(element.params.arcEndPoint.endPoint) = 0;
    MPIPathPointZ(element.params.arcEndPoint.endPoint) = 0;
```

```
    element.params.arcEndPoint.direction = 1;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    element.type = MPIPathElementTypeLINE;
    MPIPathPointX(element.params.line.point) = 0.0;
    MPIPathPointY(element.params.line.point) = 0.0;
    MPIPathPointZ(element.params.line.point) = 0.0;

    returnValue = mpiPathAppend(path,
                                &element);
    msgCHECK(returnValue);


    /* Finished appending elements to the path */

    /* Generate motion parameters from the path object */
    returnValue = mpiPathMotionParamsGenerate(path,
                                      &motionParams);
    msgCHECK(returnValue);

    /* Keep the points in memory, in case we need to backup on path */
    motionParams.bspline.point.retain = TRUE;

    /* Specify the minimum number of points required in
        the XMP buffer -- if there are less, motion will E_STOP */
    motionParams.bspline.point.emptyCount = EMPTY_COUNT;

    /* This motion will not be appended */
    motionParams.bspline.point.final = TRUE;

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MOTION_TYPE,
                       &motionParams);
    fprintf(stderr,
            "mpiMotionStart returns 0x%x: %s\n",
            returnValue,
            mpiMessage(returnValue, NULL));

    /* Collect motion events */
    while (returnValue == MPIMessageOK) {
        MPIEventStatus  eventStatus;

        /* Wait for event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               TIMEOUT);
        msgCHECK(returnValue);

        if (eventStatus.type == MPIEventTypeMOTION_DONE) {
            printf("Motion Done\n");
```

```
            break;
        }
        else {
            fprintf(stderr,
                "mpiNotifyEventWait(0x%x, 0x%x, %d) returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                notify,
                &eventStatus,
                MPIWaitFOREVER,
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        }
    }

    fprintf(stderr,
            "%s exiting: returnValue 0x%x: %s\n",
            argv[0],
            returnValue,
            mpiMessage(returnValue, NULL));

    returnValue = mpiPathDelete(path);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis[0]);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis[1]);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);
}
```

PT1.c -- Simple motion path generation, specified by position/time points.

```
/* pt1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/pt1.c 9     7/23/01 2:36p Kevinh $";
#endif

/*

:Simple motion path generation, specified by position/time points.

This sample code demonstrates the motion type, MPIMotionTypePT.  A simple
 trapezoidal velocity profile points list is created, and downloaded to
 the controller.  The time delta between each position is constant and
 the positions are spaced to generate an acceleration, constant velocity,
 and deceleration profile to the final position.

Several motion parameters must be initialized for the PT motion type:

 params.pt.pointCount - Specifies the number of points (position/time).

 params.pt.position - Pointer to a position[...] array.  There is one
    position value per point, per axis.  The length of the array must be
    equal to pointCount multiplied by the number of axes.  The positions
    are interleaved in the array by the axis index.

    For example, a three axis system would have:
        position[0] = position for axis 0
        position[1] = position for axis 1
        position[2] = position for axis 2
        position[3] = position for axis 0
        position[4] = position for axis 1
        etc.

 params.pt.time - Pointer to a time[...] array.  There is one time value
    per point.  The time specifies the number of seconds between the specified
    position, and the next position (point).  The length of the time array must
    be equal to the pointCount.

 params.pt.point.retain - Specifies whether the MPI and XMP should buffer the
    points after they have executed.  If retain = 0, the buffer will destroy
    the points after they have executed.  If retain = 1, the buffer will keep the
    points after they have executed.  This parameter is useful for future backup
```

```
    on path capability.

 params.pt.point.final - Specifies if more points will be loaded.  If final = 1,
    no more points will be loaded.  If final = 0, more points can be loaded
    with mpiMotionModify(...) using the APPEND attribute.

 params.pt.point.emptyCount - Specifies how many points the XMP-Series
    controller must have in it's buffer.  If the XMP's point buffer falls
    below the emptyCount, an E-Stop event will be generated by the motion
    supervisor, decelerating all the associated axes to a stop.

When the points are passed to mpiMotionStart(...), the MPI calculates
 constant velocity segments to fit exactly through the specified positions at
 the specified times.  The MPI library automatically handles buffering points
 lists that are larger than the XMP's buffer.  Presently, the XMP-Series
 controller can store 128 Frames per axis.  Each point requires one Frame
 per axis.

Special Considerations:

It is not necesary to specify the initial command position as the first
 position point.  The time[...] values can be constant or vary for each point.
 Constant time[...] values make the programmer's calculations easier.  For
 optimum performance, the number of points can be reduced, with no loss of
 resolution, by replacing the constant velocity sections with longer
 time[...] values.

When creating points lists longer than 64 points, you must create an Event
 Manager.  The Event Manager handles point buffering between the MPI and the XMP
 controller.

For Acceleration/Jerk fits through a points list, use the motion type,
 MPIMotionTypePVT. In this case, position/velocity/time points are specified.


Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    pt1Main

argMainRENAME(main, pt1)
#endif
```

```
#define AXIS_COUNT      (1)
#define POINT_COUNT     (25)    /* Number of points in trajectory */
#define TIME            (.5)    /* Time between points (seconds) */

double position[AXIS_COUNT * POINT_COUNT];
double ptime[POINT_COUNT];

/* Simple trapezoidal profile velocity path */
double point[POINT_COUNT] = {
    0.0,
    10.0,       /* Acceleration */
    30.0,
    60.0,
    100.0,
    150.0,
    210.0,
    280.0,
    360.0,
    450.0,
    550.0,
    650.0,      /* Constant vel */
    750.0,
    850.0,
    950.0,
    1050.0,
    1140.0,     /* Deceleration */
    1220.0,
    1290.0,
    1350.0,
    1400.0,
    1440.0,
    1470.0,
    1490.0,
    1500.0 };



/* Command line arguments and defaults */
long    axisNumber[AXIS_COUNT] = { 0,  };
long    motionNumber = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;                /* Control object */
    MPIAxis         axis[MEIXmpMAX_Axes];   /* Array of axis objects */
```

```
    MPIMotion       motion;                 /* Motion object */
    MPINotify       notify;                 /* Event notification handle */
    MPIEventMgr     eventMgr;               /* Event manager handle */
    MPIMotionParams params;                 /* MPI motion parameters */

    Service         service;                /* Event manager service handle */

    long    returnValue;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIEventMask        eventMask;

    long    argIndex;
    long    pointIndex;
    long    axisIndex;
    long    motionDone;

    double  initialPosition[MEIXmpMAX_Axes];

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
```

```
    }

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create Axis objects */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        axis[axisIndex] =
            mpiAxisCreate(control,
                          axisNumber[axisIndex]);
        msgCHECK(mpiAxisValidate(axis[axisIndex]));
    }

    /* Create motion object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        NULL);
    msgCHECK(mpiMotionValidate(motion));

    /* Append axis objects to motion object */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue =
            mpiMotionAxisAppend(motion,
                                axis[axisIndex]);
        msgCHECK(returnValue);

        /* Read initial command position */
        returnValue =
            mpiAxisCommandPositionGet(axis[axisIndex],
                                      &initialPosition[axisIndex]);
        msgCHECK(returnValue);

    }

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
```

```
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* Default (max) priority */
                      -1);  /* Default sleep (msec) */
    meiASSERT(service != NULL);

    /* Initialize motion params structure */
    params.pt.pointCount = POINT_COUNT;
    params.pt.position = position;
    params.pt.time = ptime;
    params.pt.point.retain = 0;     /* Flush frame buffer after execution */
    params.pt.point.final = 1;      /* Last point */
    params.pt.point.emptyCount = 5; /* Start E-Stop if frames left to execute
                                       is less than this limit. -1 disables */

    /* Create points */
    for (pointIndex = 0; pointIndex < POINT_COUNT; pointIndex++) {

        ptime[pointIndex] = TIME;   /* Delta time between points */

        for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
            /* Position at point */
            position[(pointIndex * AXIS_COUNT) + axisIndex] =
                point[pointIndex] + initialPosition[axisIndex];
        }
    }

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypePT,
                       &params);
    fprintf(stderr,
        "mpiMotionStart returns 0x%x: %s\n",
        returnValue,
        mpiMessage(returnValue, NULL));
    msgCHECK(returnValue);


    /* Collect motion events */
    motionDone = FALSE;
    while (motionDone != TRUE) {
```

```
        MPIEventStatus  eventStatus;

        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);

        switch(eventStatus.type) {

            case MPIEventTypeMOTION_DONE: {
                motionDone = TRUE;
                break;
            }
            default: {
                break;
            }
        }

        fprintf(stderr,
                "mpiNotifyEventWait() returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        msgCHECK(returnValue);

    }

    /* Delete Objects */
    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue = mpiAxisDelete(axis[axisIndex]);
        msgCHECK(returnValue);
    }

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

PTAppend.c -- Simple motion path generation, using position/time points with Motion Append

```
/* ptappend.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/PTAppend.c 4     7/23/01 2:36p Kevinh $";
#endif

/*

:Simple motion path generation, using position/time points with Motion Append

This sample code demonstrates the motion type, MPIMotionTypePT, with Motion
 Append.  A simple trapezoidal velocity profile points list is created, and
 downloaded to the controller.  The time delta between each position is
 constant and the positions are spaced to generate an acceleration, constant
 velocity, and deceleration profile to the final position.

Several motion parameters must be initialized for the PT motion type:

 params.pt.pointCount - Specifies the number of points (position/time).

 params.pt.position - Pointer to a position[...] array.  There is one
    position value per point, per axis.  The length of the array must be
    equal to pointCount multiplied by the number of axes.  The positions
    are interleaved in the array by the axis index.

    For example, a three axis system would have:
        position[0] = position for axis 0
        position[1] = position for axis 1
        position[2] = position for axis 2
        position[3] = position for axis 0
        position[4] = position for axis 1
        etc.

 params.pt.time - Pointer to a time[...] array.  There is one time value
    per point.  The time specifies the number of seconds between the specified
    position, and the next position (point).  The length of the time array must
    be equal to the pointCount.

 params.pt.point.retain - Specifies whether the MPI and XMP should buffer the
    points after they have executed.  If retain = 0, the buffer will destroy
    the points after they have executed.  If retain = 1, the buffer will keep the
    points after they have executed.  This parameter is useful for future backup
```

```
    on path capability.

 params.pt.point.final - Specifies if more points will be loaded.  If final = 1,
    no more points will be loaded.  If final = 0, more points can be loaded
    with mpiMotionModify(...) using the APPEND attribute.

 params.pt.point.emptyCount - Specifies how many points the XMP-Series
    controller must have in it's buffer.  If the XMP's point buffer falls
    below the emptyCount, an E-Stop event will be generated by the motion
    supervisor, decelerating all the associated axes to a stop.

When the points are passed to mpiMotionStart(...), the MPI calculates
 constant velocity segments to fit exactly through the specified positions at
 the specified times.  The MPI library automatically handles buffering points
 lists that are larger than the XMP's buffer.  Presently, the XMP-Series
 controller can store 128 Frames per axis.  Each point requires one Frame
 per axis.

Special Considerations:

It is not necesary to specify the initial command position as the first
 position point.  The time[...] values can be constant or vary for each point.
 Constant time[...] values make the programmer's calculations easier.  For
 optimum performance, the number of points can be reduced, with no loss of
 resolution, by replacing the constant velocity sections with longer
 time[...] values.

When creating points lists longer than 64 points, you must create an Event
 Manager.  The Event Manager handles point buffering between the MPI and the XMP
 controller.

For Acceleration/Jerk fits through a points list, use the motion type,
 MPIMotionTypePVT. In this case, position/velocity/time points are specified.


Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    ptappendMain

argMainRENAME(main, ptappend)
#endif
```

```
#define AXIS_COUNT      (1)
#define POINT_COUNT     (36)    /* Number of points in trajectory */
#define TIME            (0.1)   /* Time between points (seconds) */
#define POSITION_DELTA  (10.0)  /* counts for each point */

double position[AXIS_COUNT * POINT_COUNT];
double time[POINT_COUNT];

double position2[AXIS_COUNT * POINT_COUNT];
double time2[POINT_COUNT];


/* Simple trapezoidal profile velocity path */
double point[POINT_COUNT + 1];

/* Command line arguments and defaults */
long    axisNumber[AXIS_COUNT] = { 0,  };
long    motionNumber = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};


int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;                /* Control object */
    MPIAxis         axis[MEIXmpMAX_Axes];   /* Array of axis objects */
    MPIMotion       motion;                 /* Motion object */
    MPINotify       notify;                 /* Event notification handle */
    MPIEventMgr     eventMgr;               /* Event manager handle */
    MPIMotionParams params;                 /* MPI motion parameters */
    MPIMotionParams params2;                    /* MPI motion parameters */


    Service         service;                /* Event manager service handle */

    long    returnValue;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;
    long    pointIndex;
    long    axisIndex;
    long    motionDone;

    MPIEventMask    eventMask;

    double  initialPosition[MEIXmpMAX_Axes];
```

```
    double pos = 0.0;


    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create Axis objects */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        axis[axisIndex] =
            mpiAxisCreate(control,
                          axisNumber[axisIndex]);
```

```
        msgCHECK(mpiAxisValidate(axis[axisIndex]));
    }

    /* Create motion object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        NULL);
    msgCHECK(mpiMotionValidate(motion));

    /* Append axis objects to motion object */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue =
            mpiMotionAxisAppend(motion,
                                axis[axisIndex]);
        msgCHECK(returnValue);

        /* Read initial command position */
        returnValue =
            mpiAxisCommandPositionGet(axis[axisIndex],
                                      &initialPosition[axisIndex]);
        msgCHECK(returnValue);

    }

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* Default (max) priority */
                      -1);  /* Default sleep (msec) */
```

```
    meiASSERT(service != NULL);

    /* Initialize motion params structure */
    params.pt.pointCount = POINT_COUNT;
    params.pt.position = position;
    params.pt.time = time;
    params.pt.point.retain = TRUE;      /* Flush frame buffer after execution */
    params.pt.point.final = 0;      /* Last point */
    params.pt.point.emptyCount = 2; /* Start E-Stop if frames left to execute
                                       is less than this limit. -1 disables */

    /* Initialize second motion params structure */
    params2.pt.pointCount = POINT_COUNT;
    params2.pt.position = position2;
    params2.pt.time = time2;
    params2.pt.point.retain = TRUE;     /* Flush frame buffer after execution */
    params2.pt.point.final = 1;     /* Last point */
    params2.pt.point.emptyCount = 5;    /* Start E-Stop if frames left to execute
                                       is less than this limit. -1 disables */

    for(pointIndex = 0; pointIndex < (POINT_COUNT + 1); pointIndex++)
    {
        point[pointIndex] = pos;
        pos += (double) POSITION_DELTA;
    }

    /* Create points */
    for (pointIndex = 0; pointIndex < POINT_COUNT; pointIndex++) {

        time[pointIndex] = TIME;    /* Delta time between points */

        for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
            /* Position at point */
            position[(pointIndex * AXIS_COUNT) + axisIndex] =
                point[(pointIndex + 1)] + initialPosition[axisIndex];
        }
    }

    /* Create secondary points */
    for (pointIndex = 0; pointIndex < POINT_COUNT; pointIndex++) {

        time2[pointIndex] = TIME;   /* Delta time between points */

        for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
            /* Position at point */
            position2[(pointIndex * AXIS_COUNT) + axisIndex] =
                (point[(pointIndex + 1)] + point[POINT_COUNT]) +
initialPosition[axisIndex];
        }
    }

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypePT,
```

```
                    &params);
    fprintf(stderr,
        "mpiMotionStart returns 0x%x: %s\n",
        returnValue,
        mpiMessage(returnValue, NULL));
    msgCHECK(returnValue);

    /* Modify motion */
    returnValue =
        mpiMotionModify(motion,
                    MPIMotionTypePT | MPIMotionAttrMaskAPPEND,
                    &params2);
    fprintf(stderr,
        "mpiMotionModify returns 0x%x: %s\n",
        returnValue,
        mpiMessage(returnValue, NULL));
    msgCHECK(returnValue);

    /* Collect motion events */
    motionDone = FALSE;
    while (motionDone != TRUE) {

        MPIEventStatus  eventStatus;

        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);

        switch(eventStatus.type) {

            case MPIEventTypeMOTION_DONE: {
                motionDone = TRUE;
                break;
            }
            default: {
                break;
            }
        }

        fprintf(stderr,
                "mpiNotifyEventWait() returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        msgCHECK(returnValue);

    }

    /* Delete Objects */
    returnValue = serviceDelete(service);
    msgCHECK(returnValue);
```

```
    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue = mpiAxisDelete(axis[axisIndex]);
        msgCHECK(returnValue);
    }

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

pvt1.c -- Simple motion path generation, specified by position/velocity/time points.

```
/* pvt1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/pvt1.c 9     3/15/02 11:38a Erikb $";
#endif

/*

:Simple motion path generation, specified by position/velocity/time points.

This sample code demonstrates the motion type, MPIMotionTypePVT.  A simple
 trapezoidal velocity profile points list is created, and downloaded to
 the controller.  The time delta between each position is constant and
 the positions/velocities are spaced to generate an acceleration, constant
 velocity, and deceleration profile to the final position.

Several motion parameters must be initialized for the PVT motion type:

 params.pvt.pointCount - Specifies the number of points (position/vel./time).

 params.pvt.position - Pointer to a position[...] array.  There is one
    position value per point, per axis.  The length of the array must be
    equal to pointCount multiplied by the number of axes.  The positions
    are interleaved in the array by the axis index.

    For example, a three axis system would have:
        position[0] = position for axis 0
        position[1] = position for axis 1
        position[2] = position for axis 2
        position[3] = position for axis 0
        position[4] = position for axis 1
        etc.

 params.pvt.velocity - Pointer to a velocity[...] array.  There is one
    velocity value per point, per axis.  The length of the array must be
    equal to pointCount multiplied by the number of axes.  The velocities
    are interleaved in the array by the axis index.

    For example, a three axis system would have:
        velocity[0] = velocity for axis 0
        velocity[1] = velocity for axis 1
        velocity[2] = velocity for axis 2
```

```
        velocity[3] = velocity for axis 0
        velocity[4] = velocity for axis 1
        etc.

 params.pvt.time - Pointer to a time[...] array.  There is one time value
    per point.  The time specifies the number of seconds between the specified
    position/velocity, and the next position/velocity (point).  The length of
    the time array must be equal to the pointCount.

 params.pvt.point.retain - Specifies whether the MPI and XMP should buffer the
    points after they have executed.  If retain = 0, the buffer will destroy
    the points after they have executed.  If retain = 1, the buffer will keep
    the points after they have executed.  This parameter is useful for future
    backup on path capability.

 params.pvt.point.final - Specifies if more points will be loaded.  If final
    = 1, no more points will be loaded.  If final = 0, more points can be
    loaded with mpiMotionModify(...) using the APPEND attribute.

 params.pvt.point.emptyCount - Specifies how many points the XMP-Series
    controller must have in it's buffer.  If the XMP's point buffer falls
    below the emptyCount, an E-Stop event will be generated by the motion
    supervisor, decelerating all the associated axes to a stop.

When the points are passed to mpiMotionStart(...), the MPI calculates
 accleration/jerk segments to fit exactly through the specified positions at
 the specified times.  The MPI library automatically handles buffering points
 lists that are larger than the XMP's buffer.  Presently, the XMP-Series
 controller can store 128 Frames per axis.  Each point requires one Frame
 per axis.

Special Considerations:

It is not necesary to specify the initial command position as the first
 position point.  The time[...] values can be constant or vary for each point.
 Constant time[...] values make the programmer's calculations easier.  For
 optimum performance, the number of points can be reduced, with no loss of
 resolution, by replacing the constant velocity sections with longer
 time[...] values.

When creating points lists longer than 64 points, you must create an Event
 Manager.  The Event Manager handles point buffering between the MPI and the
 XMP controller.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"
```

```
#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    pvt1Main

argMainRENAME(main, pvt1)
#endif

#define AXIS_COUNT      (1)
#define POINT_COUNT     (7)
#define TIME            (.2)    /* Time between points (seconds) */

double position[AXIS_COUNT * POINT_COUNT];
double velocity[AXIS_COUNT * POINT_COUNT];
double ptime[POINT_COUNT];

/* Simple trapezoidal profile velocity path, Positions */
double pointPosition[POINT_COUNT] = {
    0.0,
    400.0,      /* Accel to position */
    1200.0,     /* Constant vel to position */
    1600.0,     /* Decel to position */
        1200.0,     /* Move back */
        400.0,
        0.0 };

double pointVelocity[POINT_COUNT] = {
    0.0,
    4000.0,     /* Accel to vel */
    4000.0,     /* Constant vel */
    0.0,        /* Decel to zero vel */
        -4000.0,    /* Move back */
        -4000.0,
        0.0 };


/* Command line arguments and defaults */
long    axisNumber[AXIS_COUNT] = { 0,  };
long    motionNumber = 0;


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                   argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
```

```
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                                MPIAxis                      axis[AXIS_COUNT],
                                MPIMotion                    *motion,
                                MPIEventMgr            *eventMgr,
                                MPINotify                    *notify,
                                Service                      *service)
{
    MPIEventMask eventMask;
        long returnValue;
        long axisIndex;


     /* Obtain a Control handle */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize the controller */
    returnValue = mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create Axis objects */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        axis[axisIndex] =
            mpiAxisCreate(*control,
                          axisNumber[axisIndex]);
        msgCHECK(mpiAxisValidate(axis[axisIndex]));
    }

    /* Create motion object */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        NULL);
    msgCHECK(mpiMotionValidate(*motion));

    /* Append axis objects to motion object */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue =
            mpiMotionAxisAppend(*motion,
                                axis[axisIndex]);
        msgCHECK(returnValue);
    }

    /* Request notification of all events from motion */
```

```
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(*motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    *notify =
        mpiNotifyCreate(eventMask,
                        *motion);
    msgCHECK(mpiNotifyValidate(*notify));

    /* Create event manager object */
    *eventMgr = mpiEventMgrCreate(*control);
    msgCHECK(mpiEventMgrValidate(*eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(*eventMgr,
                                *notify);
    msgCHECK(returnValue);

    /* Create service thread */
    *service =
        serviceCreate(*eventMgr,
                      -1,   /* Default (max) priority */
                      -1);  /* Default sleep (msec) */
    meiASSERT(service != NULL);
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl  control,
                                    MPIAxis         axis[AXIS_COUNT],
                                    MPIMotion       motion,
                                    MPINotify       notify,
                                    MPIEventMgr     eventMgr,
                                    Service service)
{
    long axisIndex;
        long returnValue;

    /* Delete objects */
    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
```

```
    msgCHECK(returnValue);

    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        returnValue = mpiAxisDelete(axis[axisIndex]);
        msgCHECK(returnValue);
    }

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

}


int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;                /* Control object */
    MPIAxis         axis[MEIXmpMAX_Axes];   /* Array of axis objects */
    MPIMotion       motion;                 /* Motion object */
    MPINotify       notify;                 /* Event notification handle */
    MPIEventMgr     eventMgr;               /* Event manager handle */
    MPIMotionParams params;                 /* MPI motion parameters */

    Service         service;                /* Event manager service handle */

    long    returnValue;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    pointIndex;
    long    axisIndex;
    long    motionDone;

    double  initialPosition[MEIXmpMAX_Axes];

     /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

        /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                               axis,
                               &motion,
                               &eventMgr,
                               &notify,
                               &service);

        /* Append axis objects to motion object */
    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
```

```
    /* Read initial command position */
    returnValue =
        mpiAxisCommandPositionGet(axis[axisIndex],
                                  &initialPosition[axisIndex]);
    msgCHECK(returnValue);

}


/* Initialize motion params structure */
params.pvt.pointCount = POINT_COUNT;
params.pvt.position = position;
params.pvt.velocity = velocity;
params.pvt.time = ptime;
params.pvt.point.retain = 0;    /* Flush frame buffer after execution */
params.pvt.point.final = 1;     /* Last point */
params.pvt.point.emptyCount = 5;/* Start E-Stop if frames left to execute
                                   is less than this limit. -1 disables */

/* Create points */
for (pointIndex = 0; pointIndex < POINT_COUNT; pointIndex++) {

    ptime[pointIndex] = TIME;   /* Delta time between points */

    for (axisIndex = 0; axisIndex < AXIS_COUNT; axisIndex++) {
        /* Position at point */
        position[(pointIndex * AXIS_COUNT) + axisIndex] =
            pointPosition[pointIndex] + initialPosition[axisIndex];

        /* Velocity at point */
        velocity[(pointIndex * AXIS_COUNT) + axisIndex] =
            pointVelocity[pointIndex];
    }
}

/* Start motion */
returnValue =
    mpiMotionStart(motion,
                   MPIMotionTypePVT,
                   &params);
fprintf(stderr,
    "mpiMotionStart returns 0x%x: %s\n",
    returnValue,
    mpiMessage(returnValue, NULL));
msgCHECK(returnValue);


/* Collect motion events */
motionDone = FALSE;
while (motionDone != TRUE) {

    MPIEventStatus  eventStatus;

    returnValue =
        mpiNotifyEventWait(notify,
```

```
                                &eventStatus,
                                MPIWaitFOREVER);

        switch(eventStatus.type) {

            case MPIEventTypeMOTION_DONE: {
                motionDone = TRUE;
                break;
            }
            default: {
                break;
            }
        }

        fprintf(stderr,
                "mpiNotifyEventWait() returns 0x%x\n"
                "\teventStatus: type %d source 0x%x info 0x%x\n",
                returnValue,
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        msgCHECK(returnValue);

    }

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(control,
                axis,
                motion,
                notify,
                eventMgr,
                service);

    return ((int)returnValue);
}
```

quickStart1.c -- Simple point to point motion program for Quick Start procedure.

```
/* quickStart1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/quickStart1.c 6     8/01/01 2:08p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    motion1Main

argMainRENAME(main, motion1)
#endif

/*

:Simple point to point motion program for Quick Start procedure.

This is a simple program to demonstrate how to start a trapezoidal profile
 motion on a single axis.  There is a minimal error checking in this sample.

This program presumes the controller is configured so the motors can move
 safely.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define MOTION_NUMBER   (0)
#define AXIS_NUMBER     (0)

#define END_POSITION    (1000)
#define VELOCITY        (1000)
#define ACCELERATION    (5000)
```

```
#define DECELERATION    (5000)


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber)
{
    long           returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,       /* motion supervisor number */
                        *axis);             /* axis object handle */
    msgCHECK(mpiMotionValidate(*motion));
```

```
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis)
{
    long    returnValue;


    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    /* Delete axis object */
    returnValue =
        mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Display positions while waiting for the motion to be done */
void displayPositionsUntilMotionDone(MPIMotion   motion)
{
    MPIStatus   status;
    double      actual;
    double      command;
    long        motionDone;
    long        returnValue;


    /* Poll status until motion done */
    motionDone = FALSE;
    while (motionDone == FALSE) {

        /* Get the motion supervisor status */
        returnValue =
            mpiMotionStatus(motion,
                            &status,
                            NULL);
        msgCHECK(returnValue);

        /* Display position */
        mpiMotionPositionGet(motion,
                             &actual,
                             &command);
        msgCHECK(returnValue);
        printf("Axis[0] Positions: Cmd %11.3lf\tAct %11.3lf\r",
            command,
            actual);
```

```
        switch (status.state) {
            case MPIStateSTOPPING:
            case MPIStateMOVING: {
                /* Sleep for 20ms and give up control to other threads */
                meiPlatformSleep(20);
                break;
            }
            case MPIStateIDLE:
            case MPIStateERROR:
            case MPIStateSTOPPING_ERROR: {
                /* Motion is done */
                motionDone = TRUE;
                break;
            }
            default: {
                /* Unknown State */
                fprintf(stderr, "Unknown state from mpiMotionStatus.\n");
                msgCHECK(MPIMessageFATAL_ERROR);
                break;
            }
        }
    }
}


/* Perform a trapezoidal move */
void trapMove(MPIMotion      motion,
              MPITrajectory *trajectory,
              double         position)
{
    MPIMotionParams     params;         /* Motion parameters */
    MPIStatus           status;         /* Status handle */

    long returnValue;


    params.trapezoidal.trajectory   = trajectory;
    params.trapezoidal.position     = &position;

    /* Read motion status */
    returnValue =
        mpiMotionStatus(motion,
                        &status,
                        NULL);
    msgCHECK(returnValue);

    switch (status.state)
    {
        case MPIStateIDLE: {
            /* Start motion */
            returnValue =
                mpiMotionStart(motion,
                               MPIMotionTypeTRAPEZOIDAL,
                               &params);
            msgCHECK(returnValue);
```

```
            break;
        }
        case MPIStateSTOPPING:
        case MPIStateMOVING: {
            /* Modify the executing motion profile */
            returnValue =
                mpiMotionModify(motion,
                                MPIMotionTypeTRAPEZOIDAL,
                                &params);
            msgCHECK(returnValue);
            break;
        }
        case MPIStateERROR: {
            /* Error State */
            msgCHECK(MPIMotionMessageERROR);
            break;
        }
        default: {
            /* Unknown State */
            fprintf(stderr, "Unknown state from mpiMotionStatus.\n");
            msgCHECK(MPIMessageFATAL_ERROR);
            break;
        }
    }
}


/* setup move parameters, poll until motion done, start move */
void commandMoves(MPIMotion  motion,
                  double     endPosition,
                  double     velocity,
                  double     acceleration,
                  double     deceleration)
{
    MPITrajectory   trajectory;     /* trajectory information */
    double          command;        /* command position */
    double          position;       /* target position  */

    long            returnValue;    /* MPI return value */


    /* Set up trajectory information */
    trajectory.velocity     = velocity;
    trajectory.acceleration = acceleration;
    trajectory.deceleration = deceleration;

    printf("Press any key to exit...\n\n");

    /* Loop until the user presses a key */
    while (meiPlatformKey(MPIWaitPOLL) < 0) {
        /* Get command position */
        returnValue =
            mpiMotionPositionGet(motion,
                                 NULL,
                                 &command);
        msgCHECK(returnValue);
```

```
        if (command != 0.0) {
            position = 0.0;

        }
        else    {
            position = endPosition;
        }

        /* Perform a trapezoidal move */
        trapMove(motion,
                 &trajectory,
                 position);

        /* Display positions while waiting for the motion to be done */
        displayPositionsUntilMotionDone(motion);
    }
    printf("\n");
}


int main(int    argc,
         char   *argv[])
{
    MPIControl          control;    /* motion controller object handle */
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;     /* motion object handle */
    MPIAxis             axis;       /* axis object handle */


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                &axis,
                AXIS_NUMBER);

    /* Command some moves */
    commandMoves(motion,
                 END_POSITION,
                 VELOCITY,
                 ACCELERATION,
                 DECELERATION);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis);
```

```
    return MPIMessageOK;
}
```

record1.c -- Read/display data recorder records from specified axis (default 0)

```
/* record1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/record1.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Read/display data recorder records from specified axis (default 0)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    record1Main

argMainRENAME(main, record1)
#endif

#define AXIS_COUNT      (1)
#define RECORD_COUNT    (100)

/* Command line arguments and defaults */
long    axisNumber  = 0;
long    period      = 0;    /* every sample */

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-period",  ArgTypeLONG,    &period,        },

    {   NULL,       ArgTypeINVALID, NULL,   }
```

```
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIAxis     axis[AXIS_COUNT];   /* axis handle */
    MPIRecorder recorder;           /* data recorder handle */

    MPIRecorderRecord   records[RECORD_COUNT];
    MEIRecorderRecord   *recordPtr;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    recordCountRemaining;
    long    recordIndex;

    long    returnValue;    /* return value from library */

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check that the axis number is valid */
    meiASSERT((axisNumber >= 0) &&
              (axisNumber < MEIXmpMAX_Axes));

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis (%d)]\n"
                           "\t\t[-period (%d)]\n"
                           "\t\t[axisNumber ...]\n",
                           argv[0],
                           ArgUSAGE,
```

```
                        axisNumber,
                        period);
    exit(MPIMessageARG_INVALID);
}

/* Create motion controller object */
control =
    mpiControlCreate(controlType,
                     &controlAddress);
msgCHECK(mpiControlValidate(control));

/* Initialize motion controller */
returnValue = mpiControlInit(control);
msgCHECK(returnValue);

/* Create axis object */
axis[0] =
    mpiAxisCreate(control,
                  axisNumber);
msgCHECK(mpiAxisValidate(axis[0]));

/* Create and configure recorder object */
recorder =
    mpiRecorderCreate(control);
msgCHECK(mpiRecorderValidate(recorder));

returnValue =
    mpiRecorderRecordConfig(recorder,
                            (MPIRecorderRecordType)MEIRecorderRecordTypeAXIS,
                            AXIS_COUNT,
                            axis);
msgCHECK(returnValue);

/* Start recording the axis */
returnValue =
    mpiRecorderStart(recorder,
                     RECORD_COUNT,
                     period,    /* period (milliseconds) */
                     0);
msgCHECK(returnValue);

/* Fill the record buffer */
for (recordIndex = 0,
     recordCountRemaining = RECORD_COUNT; recordCountRemaining > 0; ) {
    long    countGet;

    returnValue =
        mpiRecorderRecordGet(recorder,
                             recordCountRemaining,
                             &records[recordIndex],
                             &countGet);
    msgCHECK(returnValue);

    recordCountRemaining -= countGet;
    recordIndex += countGet;
```

```
    }

    /* Stop recording */
    returnValue = mpiRecorderStop(recorder);
    if (returnValue == MPIRecorderMessageSTOPPED) {
        returnValue = MPIMessageOK;
    }
    msgCHECK(returnValue);

    /* Output record buffer to screen */
    for (recordIndex = 0; recordIndex < RECORD_COUNT; recordIndex++) {
        printf("record[%d]:\n",
                recordIndex);

        /* Cast records to MEIRecorderRecord pointer */
        recordPtr = (MEIRecorderRecord *)&records[recordIndex];

        printf("axis %d sample %d\tcommand %d\tactual %d\tdac %.4f\n",
                axisNumber,
                recordPtr->axis[0].sample,
                recordPtr->axis[0].command,
                recordPtr->axis[0].actual,
                recordPtr->axis[0].dac);
    }

    returnValue = mpiRecorderDelete(recorder);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisDelete(axis[0]);
    msgCHECK(returnValue);

    returnValue =
        mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

record2.c -- Record Data and Demonstrate How to Start/Stop/Restart Recorder

```
/* record2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/record2.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Record Data and Demonstrate How to Start/Stop/Restart Recorder

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    record2Main

argMainRENAME(main, record2)
#endif

/* Command line arguments and defaults */
long    period      = 0;    /* every sample */
long    recordCount = 100;

Arg argList[] = {
    {   "-period",  ArgTypeLONG,    &period,        },
    {   "-records", ArgTypeLONG,    &recordCount,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};
```

```
long
    recorderMode(MPIRecorder    recorder,
                 long           mode,
                 long           recordCount,
                 long           period);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;                /* motion controller handle */
    MPIRecorder recorder;               /* data recorder handle */

    MEIXmpData  *firmware;

    long    axisCount;
    long    axisNumber[MEIXmpMAX_Axes]; /* axis numbers */

    long    pointCount;
    void    *point[MEIXmpMaxRecSize];

    MPIRecorderRecord   *record;
    register MPIRecorderRecord  *recordPtr;

    long    returnValue;    /* return value from library */

    long    index;

    long    recordCountRemaining;
    long    recordIndex;
    long    recordsRead;

    long    mode;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
```

```
        }
        else {
            argIndex = argIndexNew;
        }
    }

    if (argIndex >= argc) {
        axisCount       = 1;
        axisNumber[0]   = 0;
    }
    else {
        axisCount = argc - argIndex;

        if (axisCount > MEIXmpMAX_Axes) {
            axisCount = MEIXmpMAX_Axes;
        }

        for (index = 0; index < axisCount; index++) {
            axisNumber[index] = meiPlatformAtol(argv[argIndex++]);
            meiASSERT((axisNumber[index] >= 0) &&
                      (axisNumber[index] < MEIXmpMAX_Axes));
        }
    }

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-period (%d)]\n"
                           "\t\t[-records (%d)]\n"
                           "\t\t[axisNumber ...]\n",
                           argv[0],
                           ArgUSAGE,
                           period,
                           recordCount);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    returnValue =
        mpiControlMemory(control,
                         (void **)&firmware,
                         NULL);
    msgCHECK(returnValue);

    pointCount = 0;
```

```
    point[pointCount++] = &firmware->SystemData.SampleCounter;

    for (index = 0; index < axisCount; index++) {
        point[pointCount++] = &firmware->MS[axisNumber[index]].Status;
        point[pointCount++] = &firmware->Axis[axisNumber[index]].Status;
        meiASSERT(pointCount <= MEIXmpMaxRecSize);
    }

    record =
        (MPIRecorderRecord *)meiPlatformAlloc(sizeof(*record) *
                                              recordCount);
    meiASSERT(record != NULL);

    recorder =
        mpiRecorderCreate(control);
    msgCHECK(mpiRecorderValidate(recorder));

    returnValue =
        mpiRecorderRecordConfig(recorder,
                                MPIRecorderRecordTypePOINT,
                                pointCount,
                                point);
    msgCHECK(returnValue);

    recordPtr = record;

    mode = 0;

    meiPlatformConsole("Press any key to start recording, ESC to quit\n");

    recordCountRemaining = recordCount;

    while (recordCountRemaining > 0) {
        mode =
            recorderMode(recorder,
                         mode,
                         recordCountRemaining,
                         period);

        if (mode > 0) {
            long    countGet;

            returnValue =
                mpiRecorderRecordGet(recorder,
                                     recordCountRemaining,
                                     recordPtr,
                                     &countGet);
            msgCHECK(returnValue);

            recordCountRemaining -= countGet;
            recordPtr += countGet;
        }
        else if (mode < 0) {
            break;
```

```
        }
    }

    recordsRead = recordPtr - record;

    for (recordIndex = 0,
            recordPtr = record;
         recordIndex < recordsRead;
         recordIndex++,
             recordPtr++) {
        long    pointIndex;

        pointIndex = 0;

        printf("record[%d]: sample %d\n",
                recordIndex,
                recordPtr->point[pointIndex++]);

        for (index = 0; index < axisCount; index++) {
            printf("%d: motion status 0x%x axis status 0x%x\n",
                    axisNumber[index],
                    recordPtr->point[pointIndex],
                    recordPtr->point[pointIndex + 1]);

            pointIndex += 2;
        }
        meiASSERT(pointIndex == pointCount);
    }

    returnValue = mpiRecorderDelete(recorder);
    msgCHECK(returnValue);

    returnValue =
        meiPlatformFree(record,
                        (sizeof(*record) * recordCount));
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long
    recorderMode(MPIRecorder    recorder,
                 long           mode,
                 long           recordCount,
                 long           period)
{
    long    returnValue;

    long    modeNew;

    long    key;
```

```
    key = meiPlatformKey(MPIWaitPOLL);

    if (key <= 0) {
        if (key == 0) {
            key = meiPlatformKey(MPIWaitPOLL);
        }
        modeNew = mode;
    }
    else if (key == 0x1b) {
        modeNew = -1;
    }
    else {
        modeNew = (mode <= 0) ? 1 : 0;
    }

    if (modeNew != mode) {
        if (modeNew <= 0) {
            if (mode > 0) {
                returnValue =
                    mpiRecorderStop(recorder);
                msgCHECK(returnValue);
            }
        }
        else {
            returnValue =
                mpiRecorderStart(recorder,
                                 recordCount,
                                 period,    /* period (milliseconds) */
                                 0);
            msgCHECK(returnValue);
        }

        if (modeNew >= 0) {
            meiPlatformConsole("Press any key to %s recording, ESC to quit\n",
                               (modeNew == 0) ? "start" : "stop ");
        }
    }

    return (modeNew);
}
```

record3.c -- Interrupt-driven display of data recorder records from specified axis (default 0)

```
/* record3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/record3.c 12    7/23/01 2:36p Kevinh $";
#endif

/*

:Interrupt-driven display of data recorder records from specified axis (default 0)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/


#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    record3Main

argMainRENAME(main, record3)
#endif

/* Command line arguments and defaults */
long    period      = 0;    /* every sample */
long    recordCount = 100;
long    fullCount   = -1;

Arg argList[] = {
    {   "-period",  ArgTypeLONG,    &period,        },
    {   "-records", ArgTypeLONG,    &recordCount,   },
    {   "-full",    ArgTypeLONG,    &fullCount,     },

    {   NULL,       ArgTypeINVALID, NULL,   }
};
```

```
int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;                /* motion controller handle */
    MPIAxis     axis[MEIXmpMAX_Axes];   /* axis handles */
    MPIRecorder recorder;               /* data recorder handle */
    MPINotify   notify;                 /* event notification handle */
    MPIEventMgr eventMgr;               /* event manager handle */

    MPIEventMask    eventMask;

    Service     service;    /* service thread */

    long        axisCount;
    long        axisNumber[MEIXmpMAX_Axes]; /* axis numbers */

    MPIRecorderStatus   recorderStatus;
    MPIRecorderRecord   *record;
    register MEIRecorderRecord  *recordPtr;

    long    returnValue;    /* return value from library */

    long    index;

    long    recordingDone;

    long    recordIndex;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }
```

```
    if (argIndex >= argc) {
        axisCount       = 1;
        axisNumber[0]   = 0;
    }
    else {
        axisCount = argc - argIndex;

        if (axisCount > MEIXmpMAX_Axes) {
            axisCount = MEIXmpMAX_Axes;
        }

        for (index = 0; index < axisCount; index++) {
            axisNumber[index] = meiPlatformAtol(argv[argIndex++]);
            meiASSERT((axisNumber[index] >= 0) &&
                      (axisNumber[index] < MEIXmpMAX_Axes));
        }
    }

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-period (%d)]\n"
                           "\t\t[-records (%d)]\n"
                           "\t\t[-full (%d)]\n"
                           "\t\t[axisNumber ...]\n",
                           argv[0],
                           ArgUSAGE,
                           period,
                           recordCount,
                           fullCount);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    for (index = 0; index < axisCount; index++) {
        axis[index] =
            mpiAxisCreate(control,
                          axisNumber[index]);
        msgCHECK(mpiAxisValidate(axis[index]));
    }


    /* Create and configure recorder object */
    recorder =
        mpiRecorderCreate(control);
    msgCHECK(mpiRecorderValidate(recorder));

    returnValue =
```

```
    mpiRecorderRecordConfig(recorder,
                            (MPIRecorderRecordType)MEIRecorderRecordTypeAXIS,
                            axisCount,
                            axis);
msgCHECK(returnValue);

returnValue =
    mpiRecorderStatus(recorder,
                      &recorderStatus,
                      NULL);
msgCHECK(returnValue);

if (fullCount < 0) {
    fullCount =
        recorderStatus.recordCountMax -
            (recorderStatus.recordCountMax / 4);
}

/* Allocate memory for record buffer */
record =
    (MPIRecorderRecord *)meiPlatformAlloc(sizeof(*record) *
                                          recordCount);
meiASSERT(record != NULL);

/* Request notification of all events from recorder */
mpiEventMaskCLEAR(eventMask);
mpiEventMaskRECORDER(eventMask);    /* macro */
returnValue =
    mpiRecorderEventNotifySet(recorder,
                              eventMask,
                              NULL);
msgCHECK(returnValue);

/* Create event notification object for recorder */
notify =
    mpiNotifyCreate(eventMask,
                    recorder);
msgCHECK(mpiNotifyValidate(notify));

/* Create event manager object */
eventMgr =
    mpiEventMgrCreate(control);
msgCHECK(mpiEventMgrValidate(eventMgr));

/* Add notify to event manager's list */
returnValue =
    mpiEventMgrNotifyAppend(eventMgr,
                            notify);
msgCHECK(returnValue);

/* Create service thread */
service =
    serviceCreate(eventMgr,
                  -1,   /* default (max) priority */
                  -1);  /* -1 => enable interrupts */
meiASSERT(service != NULL);
```

```
    /* Start Recorder and wait for Event */
    returnValue =
        mpiRecorderStart(recorder,
                         recordCount,
                         period,    /* period (milliseconds) */
                         fullCount);
    msgCHECK(returnValue);

    recordPtr = (MEIRecorderRecord *)record;
    recordingDone = FALSE;
    while (recordingDone == FALSE) {
        MPIEventStatus  eventStatus;

        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);
        msgCHECK(returnValue);

        switch (eventStatus.type) {
            case MPIEventTypeRECORDER_FULL:
            case MPIEventTypeRECORDER_DONE: {
                long    countGet;

                returnValue =
                    mpiRecorderRecordGet(recorder,
                                         recordCount,
                                         (MPIRecorderRecord *)recordPtr,
                                         &countGet);
                msgCHECK(returnValue);

                recordPtr += countGet;

                if (eventStatus.type == MPIEventTypeRECORDER_DONE) {
                    recordingDone = TRUE;
                }
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }
    }

    returnValue = mpiRecorderStop(recorder);
    meiASSERT(returnValue == MPIRecorderMessageSTOPPED);

    /* Ouput all records to console */
    for (recordIndex = 0,
            recordPtr = (MEIRecorderRecord *)record;
         recordIndex < recordCount;
         recordIndex++,
             recordPtr++) {
        printf("record[%d]:\n",
```

```
                recordIndex);

        for (index = 0; index < axisCount; index++) {
            printf("axis %d sample %d\tcommand %d\tactual %d\tdac %.4f\n",
                    axisNumber[index],
                    recordPtr->axis[index].sample,
                    recordPtr->axis[index].command,
                    recordPtr->axis[index].actual,
                    recordPtr->axis[index].dac);
        }
    }

    /* Delete recorder */
    returnValue = mpiRecorderDelete(recorder);
    msgCHECK(returnValue);

    /* Free recorder buffer */
    returnValue =
        meiPlatformFree(record,
                        (sizeof(*record) * recordCount));
    meiASSERT(returnValue == MPIMessageOK);

    /* Delete remaining objects */
    for (index = 0; index < axisCount; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

record4.c -- Perform a diagonal two-axis motion and record command velocity and status.

```
/* record4.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/record4.c 15    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a diagonal two-axis motion and record command velocity and status.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    record4Main

argMainRENAME(main, record4)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;
long            period          = 0;    /* every sample */
long            recordCount     = 100;

Arg argList[] = {
```

```
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },
    {   "-period",  ArgTypeLONG,    &period,        },
    {   "-records", ArgTypeLONG,    &recordCount,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 200000.0, 200000.0,   },
    { 0.0,      0.0,        },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 100000.0,     1000000.0,  1000000.0,  50.0,   },
        { 100000.0,     1000000.0,  1000000.0,  50.0,   },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 100000.0,     1000000.0,  1000000.0,  50.0,   },
        { 100000.0,     1000000.0,  1000000.0,  50.0,   },
    },
};

/* motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIRecorder recorder;   /* data recorder handle */

    MPIAxis     axisX;      /* X axis */
    MPIAxis     axisY;      /* Y axis */
    MPIMotion   motion;     /* motion object */

    long    returnValue;    /* return value from library */
```

```
    long    index;

    MEIXmpData  *firmware;

    long    axisCount;

    long    pointCount;
    void    *point[MEIXmpMaxRecSize];

    MPIRecorderRecord   *record;
    register MPIRecorderRecord  *recordPtr;

    long    recordCountRemaining;
    long    recordIndex;
    long    recordsRead;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n"
                           "\t\t[-period (%d)]\n"
```

```
                            "\t\t[-records (%d)]\n",
                             argv[0],
                             ArgUSAGE,
                             MEIXmpMAX_Axes - AXIS_COUNT,
                             MEIXmpMAX_MSs - 1,
                             MEIMotionTypeLAST - 1,
                             period,
                             recordCount);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    returnValue =
        mpiControlMemory(control,
                         (void **)&firmware,
                         NULL);
    msgCHECK(returnValue);

    /* Create X axis object using axisNumber on controller*/
    axisX =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axisX));

    /* Create Y axis object using axisNumber + 1 on controller */
    axisY =
        mpiAxisCreate(control,
                      axisNumber[1]);
```

```
    msgCHECK(mpiAxisValidate(axisY));

    /* Create motion object */
    /* Append X axis to motion */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axisX);
    msgCHECK(mpiMotionValidate(motion));

    /* Append Y axis to motion */
    returnValue =
        mpiMotionAxisAppend(motion,
                            axisY);
    msgCHECK(returnValue);

    axisCount   = AXIS_COUNT;
    pointCount  = 0;

    point[pointCount++] = &firmware->SystemData.SampleCounter;

    for (index = 0; index < axisCount; index++) {
        point[pointCount++] = &firmware->Axis[index].Status;
        point[pointCount++] = &firmware->Axis[index].TC.Velocity;
        meiASSERT(pointCount <= MEIXmpMaxRecSize);
    }

    record =
        (MPIRecorderRecord *)meiPlatformAlloc(sizeof(*record) *
                                              recordCount);
    meiASSERT(record != NULL);

    recorder =
        mpiRecorderCreate(control);
    msgCHECK(mpiRecorderValidate(recorder));

    returnValue =
        mpiRecorderRecordConfig(recorder,
                                MPIRecorderRecordTypePOINT,
                                pointCount,
                                point);
    msgCHECK(returnValue);

    recordPtr = record;

    recordCountRemaining = recordCount;

    returnValue =
        mpiRecorderStart(recorder,
                         recordCount,
                         period,    /* period (milliseconds) */
                         0);
    msgCHECK(returnValue);
```

```
    /* Loop repeatedly */
    index = 0;
    {
        long    motionDone;

        if (returnValue == MPIMessageOK) {
            MPIMotionParams motionParams;

            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }

            returnValue =
                mpiMotionStart(motion,
                               motionType,
                               &motionParams);

            if (returnValue != MPIMessageOK) {
                printf("mpiMotionStart(0x%x, %d, 0x%x) returns 0x%x: %s\n",
                       motion,
                       motionType,
                       &motionParams,
                       returnValue,
                       mpiMessage(returnValue, NULL));
            }
        }

        while (recordCountRemaining > 0) {
            long    countGet;

            returnValue =
                mpiRecorderRecordGet(recorder,
                                     recordCountRemaining,
                                     recordPtr,
                                     &countGet);
            msgCHECK(returnValue);

            recordCountRemaining -= countGet;
            recordPtr += countGet;
```

```
        }

        /* Poll status until motion done */
        motionDone = FALSE;
        while (motionDone == FALSE) {
            MPIStatus   status;

            returnValue =
                mpiMotionStatus(motion,
                                &status,
                                NULL);
            msgCHECK(returnValue);

            switch (status.state) {
                case MPIStateMOVING: {
                    break;
                }
                case MPIStateIDLE: {
                    motionDone = TRUE;
                    break;
                }
                case MPIStateERROR: {
                    motionDone = TRUE;

                    returnValue =
                        mpiMotionAction(motion,
                                        MPIActionRESET);
                    msgCHECK(returnValue);

                    /* Wait for reset to take effect */
                    meiPlatformSleep(100);

                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }
        }

        if (++index >= MOTION_COUNT) {
            index = 0;
        }
    }

    recordsRead = recordPtr - record;

    printf("Record\tSample\tStat 0\tVel 0\tStat 1\tVel 1\n");

    for (recordIndex = 0,
            recordPtr = record;
         recordIndex < recordsRead;
         recordIndex++,
```

```
            recordPtr++) {

        printf("%d\t%d\t0X%X\t%f\t0X%X\t%f\n",
                recordIndex,
                recordPtr->point[0],
                recordPtr->point[1],
                (float)*((float *)&recordPtr->point[2]),
                recordPtr->point[3],
                (float)*((float *)&recordPtr->point[4]));

    }

    returnValue = mpiRecorderDelete(recorder);
    msgCHECK(returnValue);

    returnValue =
        meiPlatformFree(record,
                        (sizeof(*record) * recordCount));
    msgCHECK(returnValue);


    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisY);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axisX);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

scclose.c -- Transition control from open-loop to closed-loop mode.

```
/* ScClose.C */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Transition control from open-loop to closed-loop mode.

On all motors specified by MOTOR_COUNT
 and set origin on axes specified by AXIS_COUNT (note - assumes a linear
 map of motors and axes).

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

 */

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main ScClose

argMainRENAME(main, ScClose)
#endif

/* Command line arguments and defaults */
long axisNumber  = 0;
long motorNumber = 0;

void SetPositionZero(MPIAxis axis);

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },
```

```
    {   NULL,       ArgTypeINVALID, NULL,   }
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;/*  motion controller handle  */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIAxis     axis;   /* axis handle */
    MPIMotor    motor;  /* motor handle */

    MEIMotorConfig  motorConfigXmp;

    long    returnValue;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber  >= MEIXmpMAX_Axes) ||
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
```

```
                        ArgUSAGE,
                        MEIXmpMAX_Axes - 1,
                        MEIXmpMAX_Motors - 1);
    exit(MPIMessageARG_INVALID);
}

printf("Going to closed loop (zeroing position) ...");

/* Create device driver motion controller object */
control =
    mpiControlCreate(controlType,
                     &controlAddress);
msgCHECK(mpiControlValidate(control));

/* Initialize motion controller */
returnValue = mpiControlInit(control);
msgCHECK(returnValue);

/* Create axis objects and zero position */

axis =
    mpiAxisCreate(control,
                  axisNumber);
msgCHECK(mpiAxisValidate(axis));

SetPositionZero(axis);

/* Create motor objects and transition to closed-loop control */

motor =
    mpiMotorCreate(control,
                   motorNumber);
msgCHECK(mpiMotorValidate(motor));

returnValue =
    mpiMotorConfigGet(motor,
                      NULL,
                      &motorConfigXmp);
msgCHECK(returnValue);

motorConfigXmp.Commutation.Mode = MEIXmpCommModeCLOSED_LOOP;

returnValue =
    mpiMotorConfigSet(motor,
                      NULL,
                      &motorConfigXmp);
msgCHECK(returnValue);

/* Delete objects */
returnValue = mpiMotorDelete(motor);
msgCHECK(returnValue);
```

```
    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    meiASSERT(returnValue == MPIMessageOK);

    return ((int)returnValue);
}

void SetPositionZero(MPIAxis axis)
{
    double  actualPosition;
    double  origin;

    long    returnValue;

    returnValue =
        mpiAxisActualPositionGet(axis,
                                 &actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    returnValue =
        mpiAxisOriginGet(axis,
                         &origin);
    meiASSERT(returnValue == MPIMessageOK);

    actualPosition += origin;

    returnValue =
        mpiAxisOriginSet(axis,
                         actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    meiPlatformSleep(15);

    actualPosition = 0;

    returnValue =
        mpiAxisCommandPositionSet(axis,
                                  actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    meiPlatformSleep(15);
};
```

scdither.c -- Sinusoidal Commutation Initialization Using the Dithering Method

```
/* ScDither.C   */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

: Sinusoidal Commutation Initialization Using the Dithering Method

Uses Dithering initialization for one axis. Axis remains in open-loop
  mode with amplifier enabled.

Note -
  #define provided for reversing encoder and DAC phasing. Program disables
  HW Limits, Amp Fault, Home and Error Limit.

The commutation configuration parameters will be different
  for your application.  Take particular care in setting the open-loop
  continuous DAC OutputLevel as this must be a safe level for your specific
  drive and motor combination.  Please read Sinusoidal Commutation Documentation
  in the MPI/MEI Software Reference Manual for further information.

Successful initialization with "Dithering" requires a system with minimal
  influence from external forces (e.g., gravitational force, cable carrier, etc)
  and static friction.

The parameters DAC level ("OutputLevel") and time delay ("VEL_DELAY" and
  "ACC_DELAY")  should be set for optimum performance with the user's specific
system.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    ScDither

argMainRENAME(main, ScDither)
#endif
```

```
#define VEL_DELAY           (5)                     /* servo samples */
#define ACC_DELAY           (5)                     /* servo samples */
#define RECORD_COUNT_MAX    (VEL_DELAY + ACC_DELAY) /* number of data recorder points
*/

/* Command line arguments and defaults */
long    axisNumber      = 0;
long    motionNumber    = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Option to reverse encoder phasing */
#undef ReverseEncPhasing
/* Option to reverse DAC phasing */
#undef ReverseDACPhasing

MEIRecorderRecord   Record[RECORD_COUNT_MAX];

double
    accelActualGet(MPIRecorder  recorder,
                   MPIAxis      axis);

void
    setPositionZero(MPIAxis axis);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis handles */
    MPIFilter   filter;     /* filter handles */
    MPIMotor    motor;      /* motor handles  */
    MPIMotion   motion;     /* coordinated motion object handles */
    MPIRecorder recorder;   /* data recorder handle */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIControlConfig    controlConfig;      /* motion controller configuration */
    MPIAxisConfig       axisConfig;         /* axis configuration */
    MPIMotorConfig      motorConfig;
    MEIMotorConfig      motorConfigXmp;
    MPIFilterGain       gain;
    MEIFilterGainPID    *pid;

    MEIXmpData  *firmware;

    MEIXmpCommutationBlock  *commutation;

    long    nEncCountPerCycle = 4096; /* number of encoder counts per motor
revolution */
```

```
    float   OpenDAC_Level   =   (float)3277.0;  /* Open Loop DAC Level where 32767. =
10V) */
    float   N_CYCLES    =       (float)3.0;     /* number of elec. cycles per motor
rev. */

    long    returnValue;    /* return value from library */

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    returnValue =
        mpiControlMemory(control,
                         (void**)&firmware,
```

```
                        (void**)NULL);
    msgCHECK(returnValue);

    /*  Create a Data Recorder for future use   */
    recorder =
        mpiRecorderCreate(control);
    msgCHECK(mpiRecorderValidate(recorder));

    /*
        Configure motion controller
        Set Sample Rate = 2 kHz
        configure for commutation B phase DACs and ADCs
     */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    controlConfig.sampleRate = 2000;

    /*
        When supporting commutation,
        Cmd and Aux DACs must be enabled.
        Enabling the same number of Cmd and Aux DACs as there are motors.
     */
    controlConfig.cmdDacCount = controlConfig.motorCount;
    controlConfig.auxDacCount = controlConfig.motorCount;

    /*  Configure required number of A/Ds   */
    controlConfig.adcCount = 8;

    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    /*  Create axis, filter and motor controller objects    */

    /*  Create axis */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /*  Create filter   */
    filter =
        mpiFilterCreate(control,
                        axisNumber);
    msgCHECK(mpiFilterValidate(filter));

    /*  Create motor    */
    motor =
        mpiMotorCreate(control,
                       axisNumber);
    msgCHECK(mpiMotorValidate(motor));
```

```
    /*  Disable the amplifier   */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                             FALSE);
    msgCHECK(returnValue);

    /*  Configure axis  */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    /*  Set inPositionLimit */
    axisConfig.inPosition.tolerance.positionFine = (float)1.0e10;

    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    /*  Configure Motor */
    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /*
        Disable limit error, home, amp fault & H/W Limits
        Set encoder phase
     */

    /*  Disable limit error */
    motorConfig.event[MPIEventTypeLIMIT_ERROR].action           = MPIActionNONE;
    motorConfig.event[MPIEventTypeLIMIT_ERROR].trigger.error    = (float)50000;

    /*  Disable home event  */
    motorConfig.event[MPIEventTypeHOME].action  = MPIActionNONE;

    /*  Disable amp fault event */
    motorConfig.event[MPIEventTypeAMP_FAULT].action = MPIActionNONE;

    /*  Disable neg. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_NEG].action  = MPIActionNONE;

    /*  Disable pos. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_POS].action  = MPIActionNONE;

    /*
            Set encoder phase:

            0 - for normal
            1 - for reversed
     */
#if defined(ReverseEncPhasing)
    motorConfig.encoderPhase = 1;
```

```
#else
    motorConfig.encoderPhase = 0;
#endif

    /* Setup commutation  */
    commutation = &motorConfigXmp.Commutation;

    /* Motor Controller commutation parameters */
    commutation->Mode        = MEIXmpCommModeOPEN_LOOP;
    commutation->Length      = nEncCountPerCycle;
    commutation->Scale       = (float)MEIXmpCOMM_TABLE_SIZE * N_CYCLES /
                               (float)commutation->Length;
    commutation->PhaseDelta  = MEIXmpCOMM_120DEGREES;
    commutation->OutputLevel = OpenDAC_Level;
    commutation->Offset      = (long)0;

#if defined(ReverseDACPhasing)
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseINVERTED;
#else
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseNORMAL;
#endif


    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Get current configuration */
    returnValue =
        mpiFilterGainGet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);

    pid = (MEIFilterGainPID *)&gain;

    /* Filter Controller control parameters */
    pid->gain.proportional          = (float)50.0;
    pid->gain.integral              = (float)0.0;
    pid->gain.derivative            = (float)200.0;
    pid->feedForward.velocity       = (float)0.0;
    pid->feedForward.acceleration   = (float)0.0;
    pid->feedForward.friction       = (float)0.0;
    pid->integrationMax.moving      = (float)0.0;
    pid->integrationMax.rest        = (float)0.0;
    pid->output.limit               = (float)26213.6;
    pid->output.offset              = (float)0.0;

    /* Set filter configuration  */
    returnValue =
        mpiFilterGainSet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);
```

```
    /*
        Create a Motion Object and append an Axis
        Note 1 - separate axis appends will be required for coordinated motion
        Note 2 - Axis Map does not take effect until motion is commanded
     */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Clear out any previous errors */
    returnValue =
        mpiMotionAction(motion,
                        MPIActionRESET);
    msgCHECK(returnValue);

    /*  Immediately send Map to controller  */
    returnValue =
        mpiMotionAction(motion,
                        (MPIAction)MEIActionMAP);
    msgCHECK(returnValue);

    /*  Enable the amplifier    */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                             TRUE);
    msgCHECK(returnValue);

    /*  Begin "Dither" phase-finding based on motor acceleration sequence   */
    {
        double  angle;
        double  delta;

        long    index;

        angle = 0.0;
        delta = 256.0;

        /*
            Find Field magnetic vector by dithering Stator vector (monitor
            direction of acceleration
         */
        for (index = 0; index < 8; index++) {
            double  accel;

            long    angleSet;

            accel =
                accelActualGet(recorder,
                               axis);

            if (accel < 0.0) {
                angle += delta;
            }
            else {
                angle -= delta;
            }
```

```
            angleSet = (long)angle;

            returnValue =
                mpiMotorMemorySet(motor,
                                  &firmware->Motor[axisNumber].Commutation.Offset,
                                  &angleSet,
sizeof(firmware->Motor[axisNumber].Commutation.Offset));
            msgCHECK(returnValue);

            delta *= .75;

            /* Avoid dual null position issue with first dither */
            if ((index == 0) &&
                (accel == 0.0)) {
                delta *= 2;
            }
        }
    }

    /*  Zero Position   */
    setPositionZero(axis);

    /*  Go Closed Loop  */
#if 0
    returnValue =
        meiMotorCommutationModeSet(motor,
                                   MEIXmpCommModeCLOSED_LOOP);
    msgCHECK(returnValue);
#endif

    /*  Delete Objects  */
    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiFilterDelete(filter);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiRecorderDelete(recorder);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

double
    accelActualGet(MPIRecorder  recorder,
                   MPIAxis      axis)
{
```

```
    long    returnValue;    /* return value from library */

    MEIRecorderRecord   *recordPtr;

    long    recordsRemaining;


    long    v1;
    long    v2;

    returnValue =
        mpiRecorderRecordConfig(recorder,
                                (MPIRecorderRecordType)MEIRecorderRecordTypeAXIS,
                                1,
                                &axis);
    msgCHECK(returnValue);

    returnValue =
        mpiRecorderStart(recorder,
                         RECORD_COUNT_MAX,  /* number of records */
                         0,                 /* period (milliseconds) */
                         0);
    msgCHECK(returnValue);

    recordPtr = Record;
    recordsRemaining = RECORD_COUNT_MAX;

    while (recordsRemaining > 0) {
        long    recordsRead;

        returnValue =
            mpiRecorderRecordGet(recorder,
                                 recordsRemaining,
                                 (MPIRecorderRecord *)recordPtr,
                                 &recordsRead);
        msgCHECK(returnValue);

        recordsRemaining -= recordsRead;
        recordPtr        += recordsRead;
    }

    returnValue = mpiRecorderStop(recorder);

    if (returnValue == MPIRecorderMessageSTOPPED) {
        returnValue = MPIMessageOK;
    }
    msgCHECK(returnValue);

    v1 =
        Record[VEL_DELAY - 1].axis[0].actual -
        Record[0].axis[0].actual;

    v2 =
        Record[RECORD_COUNT_MAX - 1].axis[0].actual -
        Record[RECORD_COUNT_MAX - VEL_DELAY].axis[0].actual;

    return ((double)(v2 - v1));
}
```

```
void
    setPositionZero(MPIAxis axis)
{
    double  actualPosition;
    double  origin;

    long    returnValue;

    returnValue =
        mpiAxisActualPositionGet(axis,
                                 &actualPosition);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisOriginGet(axis,
                         &origin);
    msgCHECK(returnValue);

    actualPosition += origin;

    returnValue =
        mpiAxisOriginSet(axis,
                         actualPosition);
    msgCHECK(returnValue);

    meiPlatformSleep(15);

    actualPosition = 0;

    returnValue =
        mpiAxisCommandPositionSet(axis,
                                  actualPosition);
    msgCHECK(returnValue);

    meiPlatformSleep(15);
}
```

schall.c -- Single-Axis Sinusoidal Commutation Sample Program with Hall Initialization

```
/* ScHall.C */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software contains proprietary and confidential information of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Single-Axis Sinusoidal Commutation Sample Program with Hall Initialization

This program reads the Hall - I/O bit state, then sets the commutation
 table entry point to match the active Hall sensor.  This version latches
 position and up-dates the commutation entry point based on the first
 Hall transition (latches upon a single I/O with a specific edge).



        Axis 0:     Uses Hall-type initialization with a small trapezoidal
                    move (closed-loop) to the first Hall transition. Updates
                    commutation position based on the Hall transition location.

    Background:     The initial state of the three Hall sensors provides six
                    possible magnetic sectors of 60 degrees which may contain
                    the motor field vector (i.e., rotor magnetic position).
                    This program uses the following logic table:


 Mag.Phase Angle:    0 - 60 60 - 120 120 - 180  180 - 240 240 - 300  300 - 360
                    ---------------------------------------------------------
                    |       |       |         |          |          |         |
    Hall A Sensor   |  On   |   On  |   On    |    Off   |   Off    |    Off  |
                    |       |       |         |          |          |         |
    Hall B Sensor   |  Off  |   Off |   On    |    On    |   On     |    Off  |
                    |       |       |         |          |          |         |
    Hall C Sensor   |  On   |   Off |   Off   |    Off   |   On     |    On   |
                    ---------------------------------------------------------

 Hall Logic (C,B,A) (101)     (001)    (011)      (010)     (110)      (100)

 Hall Region           5       1        3          2         6          4


            The rotor magnetic field vector will initially be located within
            one of these six regions.  Sufficient accuracy for closed-loop control
            is obtained by assigning the magnetic location to the mid-point of
            the active region.  Thus, the actual commutation accuracy will be +/- 30
            electrical degrees (assuming the halls have been accurately located on
your
            motor).  The motor is configured under closed-loop control and a slow
```

```
            trapezoidal move is implemented to the first observed Hall transition.
            The commutation position is then corrected to reflect the exact magnetic
            position of the Hall transition point.

            Note -  The XMP calculates: A Phase DAC Output = sin(Theta + 120)
                                        B Phase DAC Output = sin(Theta)

            Note -  See the XMP Installation Manual for information on wiring the
                    Transceiver I/O to the Hall sensors.

            Note -  #defs are provided to reverse encoder and DAC phasing.  Be aware
                    that the DAC phasing change should be called only once.



  Configure the XMP:

            Axis 0  Filter 0   Motor 0


  Configure Transceivers as input (default state):

            This sample application configures three transceivers on one motor.


 Warning!   This program disables the HW Limits, Amp Fault, Home and Error Limit.

 Warning!  The commutation configuration parameters will be different
  for your application.  Take particular care in setting the open-loop
  continuous DAC OutputLevel as this must be a safe level for your specific
  drive and motor combination.  Please read the MEI Sinusoidal Commutation
  Documentation.

 Warning!  This is a sample program to assist in the integration of the
  XMP motion controller with your application.  It may not contain all
  of the logic and safety features that your application requires.

 */

 #include <stdlib.h>
 #include <stdio.h>

 #include "stdmpi.h"
 #include "stdmei.h"

 #include "apputil.h"

 #if defined(ARG_MAIN_RENAME)
 #define main ScHall

 argMainRENAME(main, ScHall)
 #endif

 #define AXIS_CNT    (1)
 #define SLOW        (5.E2)      /* velocity to first Hall transition, cts/sec */
 #define SLOW_ACCEL  (1.E4)      /* acceleration to first Hall transition cts/sec */
 #define FAST        (5.E4)      /* velocity to shift commutation pointer, cts/sec */
 #define FAST_ACCEL  (5.E5)      /* acceleration to shift comm. pointer cts/sec */
```

```
#undef USERIO

#if defined USERIO    /* Toggle the user bit associated with the motor */
#define BIT_MASK MEIXmpMotorIOMaskUSER   /* user IO Bit Mask */
#define BIT_IO   MEIXmpMotorIOBitUSER   /* user IO Bit */
#endif


/* User Settings MotorIO transceivers */
#define IO_CONFIG_IN        (MEIMotorTransceiverConfigINPUT) /* INPUT or OUTPUT   */
#define TRANSC_A_ID         (MEIMotorTransceiverIdA)
#define TRANSC_A_MASK       (MEIMotorTransceiverMaskA)
#define TRANSC_B_ID         (MEIMotorTransceiverIdB)
#define TRANSC_B_MASK       (MEIMotorTransceiverMaskB)
#define TRANSC_C_ID         (MEIMotorTransceiverIdC)
#define TRANSC_C_MASK       (MEIMotorTransceiverMaskC)


/* Command line arguments and defaults */
long    axisNumber      = 0;
long    motionNumber    = 0;
long    motorNumber     = 0;


/* Capture Parameters */
#define ACTIVE_LOW_EDGE     (0)
#define ACTIVE_HIGH_EDGE    (1)

long captureActiveEdge = ACTIVE_HIGH_EDGE;      /* capture on high edge */

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Option to reverse encoder phasing */
#undef ReverseEncPhasing
/* Option to reverse DAC phasing */
#undef ReverseDACPhasing
#undef DumpCommParameters


void SetPositionZero(MPIAxis axis);


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis handles */
    MPIFilter   filter;     /* filter handles */
    MPIMotor    motor;      /* motor handles  */
    MPICapture  capture;    /* capture handles */
```

```
    MPIControlConfig    controlConfig;  /* motion controller configuration */
    MPIAxisConfig       axisConfig;     /* axis configuration */
    MPIFilterGain       gain;
    MPIMotorConfig      motorConfig;    /* motor configuration */
    MPIMotorEventConfig eventConfig;
    MPIMotorIo          io;
    MPICaptureConfig    captureConfig;  /* capture configuration */
    MPICaptureStatus    captureStatus;  /* capture status */


    MEIMotorConfig      motorConfigXmp; /* contains transceiver configuration */
    MEIFilterGainPID    *pid;
    MEIXmpCommutationBlock  *commutation;

    MPIMotion       motion; /* Coordinated motion object handles */
    MPIMotionParams params; /* Motion parameters */
    MPITrajectory   trajectory;

    MPIControlType          controlType;
    MPIControlAddress       controlAddress;

    long    returnValue;    /* return value from library */
    long    iTranState;     /* value of the Hall transceiver word */
    long    iOffset;        /* integer value to move comm. Offset parameter */
    long    nEncCountPerCycle =  4096;          /* number of encoder counts per motor
revolution */
    float   OpenDAC_Level     = (float)3277.0;  /* Open Loop DAC Level where 32767. =
10V) */
    float   N_CYCLES          = (float)3.0;     /* number of elec. cycles per motor
rev. */

#if defined(ReverseDACPhasing)
    long    dacCount;
    long    dacNumber[MPIMotorDacCountMAX];
    long    number;
#endif

    double          command;
    double          endposition;
    double          LminusTh;
    double          origin;
    double          origLatchPosition, latchedPosition, latchPosChange;

    float   inPositionLimit = (float)1.0e10;

    long    argIndex;
    long    captureNumber;


    /* Parse command line for Control type and address */

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);
```

```
    /* Parse command line for application-specific arguments */

    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */

    if ((argIndex < argc) ||
        (axisNumber   >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }


    /* Calculate default capture number for axisNumber */
    captureNumber = (axisNumber/MEIXmpMotorsPerBlock) * MEIXmpMaxLatches +
                    (axisNumber % MEIXmpMotorsPerBlock) * 2;

    /* Create device driver motion controller object  */

    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /*  Initialize motion controller  */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /*
       Configure motion controller
       Set Sample Rate = 2 kHz (this is the default)
       Configure for commutation B phase DACs
       Configure ADCs
     */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);
```

```
    controlConfig.sampleRate = 2000;

    /*
        When supporting commutation,
        Cmd and Aux DACs must be enabled.
        Enabling the same number of Cmd and Aux DACs as there are motors.
     */
    controlConfig.cmdDacCount = controlConfig.motorCount;
    controlConfig.auxDacCount = controlConfig.motorCount;

    /* Configure required number of A/D  */
    controlConfig.adcCount = 8;

    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    /* Create axis, filter and motor controller objects */

    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    filter =
        mpiFilterCreate(control,
                      axisNumber);
    msgCHECK(mpiFilterValidate(filter));

    motor =
        mpiMotorCreate(control,
                      motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create capture object for captureNumber */
    capture =
        mpiCaptureCreate(control,
                         captureNumber);
    msgCHECK(mpiCaptureValidate(capture));


    /* Disable the amplifier */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                            FALSE);

    /* Disable capture */
    returnValue =
        mpiCaptureArm(capture,
                      FALSE);
    msgCHECK(returnValue);


    msgCHECK(returnValue);
```

```
    /* Configure axis  */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    /* Set inPositionLimit  */
    axisConfig.inPosition.tolerance.positionFine = inPositionLimit;

    returnValue =
        mpiAxisConfigSet(axis,
                        &axisConfig,
                        NULL);
    msgCHECK(returnValue);



    /* Configure Motor */
    returnValue =
        mpiMotorConfigGet(motor,
                         &motorConfig,
                         &motorConfigXmp);
    msgCHECK(returnValue);

    /* Disable limit error, home, amp fault & H/W Limits, set encoder phase */

    /* Disable limit error */
    motorConfig.event[MPIEventTypeLIMIT_ERROR].action           = MPIActionNONE;
    motorConfig.event[MPIEventTypeLIMIT_ERROR].trigger.error    = (float)50000;

    /* Disable home event */
    motorConfig.event[MPIEventTypeHOME].action  = MPIActionNONE;

    /* Disable amp fault event */
    motorConfig.event[MPIEventTypeAMP_FAULT].action = MPIActionNONE;

    /* Disable neg. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_NEG].action  = MPIActionNONE;

    /* Disable pos. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_POS].action  = MPIActionNONE;

    /* Verify that transceivers are configured for input (default) */
    motorConfigXmp.Transceiver[TRANSC_A_ID].Config = IO_CONFIG_IN;
    motorConfigXmp.Transceiver[TRANSC_B_ID].Config = IO_CONFIG_IN;
    motorConfigXmp.Transceiver[TRANSC_C_ID].Config = IO_CONFIG_IN;


    /* Set encoder phase; 0 - for normal; 1 - for reversed */

#ifdef ReverseEncPhasing
    motorConfig.encoderPhase = 1;
#else
    motorConfig.encoderPhase = 0;
#endif

    /* Setup Motor commutation in open-loop mode  */
```

```
    commutation = &motorConfigXmp.Commutation;

    commutation->Mode        = MEIXmpCommModeOPEN_LOOP;
    commutation->Length      = nEncCountPerCycle;
    commutation->Scale       = (float)MEIXmpCOMM_TABLE_SIZE * N_CYCLES /
                               (float)commutation->Length;
    commutation->PhaseDelta  = MEIXmpCOMM_120DEGREES;
    commutation->OutputLevel = OpenDAC_Level;   /* Open-loop DAC level */
    commutation->Offset      = (long)0;

#if defined(ReverseDACPhasing)
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseINVERTED;
#else
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseNORMAL;
#endif

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Get current Filter configuration */
    returnValue =
        mpiFilterGainGet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);

    pid = (MEIFilterGainPID *)&gain;

    /* Filter Controller control parameters */
    pid->gain.proportional          = (float)100.0; /* Kp must be set for your system
*/
    pid->gain.integral              = (float)0.0;   /* Ki set to zero for initial use
*/
    pid->gain.derivative            = (float)400.0; /* Kd must be set for your system
*/
    pid->feedForward.velocity       = (float)0.0;
    pid->feedForward.acceleration   = (float)0.0;
    pid->feedForward.friction       = (float)0.0;
    pid->integrationMax.moving      = (float)0.0;
    pid->integrationMax.rest        = (float)0.0;
    pid->output.limit               = (float)32767.0; /* closed-loop output limit */
    pid->output.offset              = (float)0.0;

    /* Set filter configuration  */
    returnValue =
        mpiFilterGainSet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);

    /*
       Create a Motion Object and append an Axis
       Note 1 - separate axis appends will be required for coordinated motion
       Note 2 - Axis Map does not take effect until motion is commanded!
```

```
     */
    motion =
         mpiMotionCreate(control,
                         motionNumber,
                         axis);
    msgCHECK(mpiMotionValidate(motion));


    /* Clear out any previous errors */
    returnValue =
        mpiMotionAction(motion,
                        MPIActionRESET);
    msgCHECK(returnValue);


    /* Immediately send Map to controller */
    returnValue =
        mpiMotionAction(motion,
                        (MPIAction)MEIActionMAP);
    msgCHECK(returnValue);


    /* Display commutation parameters */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);


 #ifdef DumpCommParameters


    printf("Motor %ld:\n",
           motorNumber);

    printf("\tCommutation Length before O-L move = %ld\n",
           motorConfigXmp.Commutation.Length);

    printf("\tCommutation Scale = %f\n",
           motorConfigXmp.Commutation.Scale);

    printf("\tCommutation Theta = %ld\n",
           motorConfigXmp.Commutation.Theta);

#endif

    if (motorConfigXmp.Commutation.Length != 0) {

        double  phaseAngle;
        long tableIndex;

        tableIndex =
            ((long)(motorConfigXmp.Commutation.Theta *
              motorConfigXmp.Commutation.Scale)) &
             (MEIXmpCOMM_TABLE_SIZE - 1);

        phaseAngle =
```

```
            (double)tableIndex * 360.0/
            (double)MEIXmpCOMM_TABLE_SIZE;

        LminusTh = (double)(motorConfigXmp.Commutation.Length -
              motorConfigXmp.Commutation.Theta);

        printf("\tPhase angle = %lf degrees\n",
                phaseAngle);
    }

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                        &motorConfig,
                        &motorConfigXmp);
    msgCHECK(returnValue);


    /*
       Implement an open-loop "move" (without drive enabled)to
       allow Trajectory Calculator Delta to move commutation
       Theta to the beginning of the commutation table (w/o motion).
       Therefore, if Theta != 0:

                Move distance = Length - Theta (counts)
     */

    if((long)(motorConfigXmp.Commutation.Theta *
              motorConfigXmp.Commutation.Scale) != 0)
        {

        returnValue =
            mpiAxisCommandPositionGet(axis,
                                      &command);
        msgCHECK(returnValue);

        trajectory.velocity     = FAST;
        trajectory.acceleration = FAST_ACCEL;
        trajectory.deceleration = FAST_ACCEL;

        params.trapezoidal.trajectory = &trajectory;
        endposition =
        command + LminusTh;
        params.trapezoidal.position = &endposition;

        meiPlatformSleep(25);

        /*   Start "virtual" motion  */
        returnValue =
            mpiMotionStart(motion,
                           MPIMotionTypeTRAPEZOIDAL,
                         &params);
        msgCHECK(returnValue);

        /*   Delay to allow "virtual" motion to complete  */
        meiPlatformSleep(500);

        }
```

```
    /* Zero command and actual positions */
        SetPositionZero(axis);



    /* Read and display the transceiver IO bit */

    returnValue =
        mpiMotorIoGet(motor,
                      &io);
    msgCHECK(returnValue);


/*
   At this point we are still in open-loop mode with the drive disabled.
   The combined transceiver word provides the basic logic needed to establish
   the commutation Offset (Offset range: 0 to 1023) required for the active
   Hall region.
 */

    iTranState = ((io.input & TRANSC_A_MASK) + (io.input & TRANSC_B_MASK) +
                  (io.input & TRANSC_C_MASK));

    printf("\n%s: %d\n","Trans A Input bit",(io.input & TRANSC_A_MASK));
    printf("\n%s: %d\n","Trans B Input bit",(io.input & TRANSC_B_MASK));
    printf("\n%s: %d\n","Trans C Input bit",(io.input & TRANSC_C_MASK));
    printf("\n%s: %d\n","Trans Input Word",iTranState);


/* Configure capture for Transceiver input trigger */
    returnValue =
        mpiCaptureConfigGet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);

/* Set commutation Offset and expected Hall transition */

    switch (iTranState)
    {

        case 1: /* w/CW rotation: next Hall Region 3, next Hall/Trans. edge B+ */
            {
            iOffset     = 256;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskB;
            captureConfig.trigger.pattern = MEIMotorTransceiverMaskB;
            break;
            }

        case 2: /* w/CW rotation: next Hall Region 6, next Hall/Trans. edge C+ */
            {
            iOffset     =  597;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskC;
            captureConfig.trigger.pattern = MEIMotorTransceiverMaskC;
            break;
            }
```

```
        case 3: /* w/CW rotation: next Hall Region 2, next Hall/Trans. edge A- */
            {
            iOffset     =  426;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskA;
            captureConfig.trigger.pattern = 0;
            break;
            }

        case 4: /* w/CW rotation: next Hall Region 5, next Hall/Trans. edge A+ */
            {
            iOffset     =  938;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskA;
            captureConfig.trigger.pattern = MEIMotorTransceiverMaskA;
            break;
            }

        case 5: /* w/CW rotation: next Hall Region 1, next Hall/Trans. edge C- */
            {
            iOffset     =   85;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskC;
            captureConfig.trigger.pattern = 0;
            break;
            }

        case 6: /* w/CW rotation: next Hall Region 4, next Hall/Trans. edge B- */
            {
            iOffset     =  768;
            captureConfig.trigger.mask = MEIMotorTransceiverMaskB;
            captureConfig.trigger.pattern = 0;
            break;
            }

        default:
            {
                printf("\n Incorrect read of Transceivers \n");
                exit(0);
            }
    }


    returnValue =
        mpiCaptureConfigSet(capture,
                            &captureConfig,
                            NULL);
    msgCHECK(returnValue);


    /* Configure Home Event action */
    returnValue =
        mpiMotorEventConfigGet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;     /* no event */
    eventConfig.trigger.polarity = TRUE;
```

```
    returnValue =
        mpiMotorEventConfigSet(motor,
                               MPIEventTypeHOME,
                               &eventConfig,
                               NULL);
    msgCHECK(returnValue);

    /* Arm the capture */
    returnValue =
        mpiCaptureArm(capture,
                      TRUE);
    msgCHECK(returnValue);


    /* Get motor configuration again */
    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);


    /* Setup commutation with closed-loop control and new Offset */

    commutation->Mode        = MEIXmpCommModeCLOSED_LOOP;
    commutation->Offset      = iOffset;     /* Hall updated Offset */

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Read original position for capture (origin) */

    returnValue =
        mpiAxisOriginGet(axis,
                         &origin);
    meiASSERT(returnValue == MPIMessageOK);

    origLatchPosition = origin;

    returnValue =
        mpiAxisOriginSet(axis,
                         origin);
    meiASSERT(returnValue == MPIMessageOK);


#ifdef DumpCommParameters

    /* Display commutation parameters again*/
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);
```

```
    printf("\tCommutation Length C-L = %ld\n",
           motorConfigXmp.Commutation.Length);


    printf("\tCommutation Theta = %ld\n",
           motorConfigXmp.Commutation.Theta);

    if (motorConfigXmp.Commutation.Length != 0) {

        double  phaseAngle;
        long tableIndex;

        tableIndex =
            ((long)(motorConfigXmp.Commutation.Theta *
              motorConfigXmp.Commutation.Scale)) &
             (MEIXmpCOMM_TABLE_SIZE - 1);

        phaseAngle =
             (double)tableIndex * 360.0/
             (double)MEIXmpCOMM_TABLE_SIZE;

        printf("\tPhase angle = %lf degrees\n",
                phaseAngle);
    }

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

#endif


    /* Enable the amplifier */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                             TRUE);
    msgCHECK(returnValue);

    /*
       Stage/motor will servo on position with initial (Hall)
       estimate of commutation position.
       Wait period: 0.5 seconds in this example.
     */

    meiPlatformSleep(500);


    /*
        Implement a closed-loop move to the first Hall transition.
        Upon reaching transition point, latch position and update
        commutation table entry point.

        The move is currently set for +72 elec. degrees. Since the
        commutation max. error is +/- 30 deg based on a 60 deg sector,
```

```
        this allows up to 12 electrical degrees of physical mis-alignment
        of the hall sensor.
     */

    returnValue =
        mpiAxisCommandPositionGet(axis,
                                  &command);
    msgCHECK(returnValue);

    trajectory.velocity     = SLOW;
    trajectory.acceleration = SLOW_ACCEL;
    trajectory.deceleration = SLOW_ACCEL;

    params.trapezoidal.trajectory = &trajectory;
    endposition =
        command +
        ((double)(nEncCountPerCycle) / ((double)(N_CYCLES * 5.0)));
    params.trapezoidal.position = &endposition;

    meiPlatformSleep(25);

    /* Start motion */
    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
                       &params);
    msgCHECK(returnValue);

    printf("\n Moving...(hit any key after motion completes)...\n\n");

    /* State Machine - Poll capture status, update display, etc. */
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        returnValue =
            mpiCaptureStatus(capture,
                             &captureStatus,
                             NULL);
        msgCHECK(returnValue);


        if (captureStatus.state == MPICaptureStateCAPTURED) {

            latchedPosition = captureStatus.latch[0];

            /* Re-arm position capture */
            returnValue =
                mpiCaptureArm(capture,
                              TRUE);
            msgCHECK(returnValue);
        }
    }

    latchPosChange = latchedPosition - origLatchPosition;

    printf("\n iOffset(orig.) = %d", iOffset);
    iOffset += (long)((((float)commutation->Length)/(N_CYCLES*12) -
 latchPosChange)*(commutation->Scale));
    printf("\n iOffset(final) = %d\n\n", iOffset);
```

```
    /* Get motor configuration again */
    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);


    /* Update commutation Offset based on Hall transition location */

    commutation->Offset      = iOffset;     /* Hall updated Offset */

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /*
        Delete Objects
     */
    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiCaptureDelete(capture);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiFilterDelete(filter);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}


void SetPositionZero(MPIAxis axis)
{
    double  actualPosition;
    double  origin;

    long    returnValue;

    returnValue =
        mpiAxisActualPositionGet(axis,
                                 &actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    returnValue =
        mpiAxisOriginGet(axis,
```

```
                        &origin);
    meiASSERT(returnValue == MPIMessageOK);

    actualPosition += origin;

    returnValue =
        mpiAxisOriginSet(axis,
                        actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    meiPlatformSleep(15);

    actualPosition = 0;

    returnValue =
        mpiAxisCommandPositionSet(axis,
                                  actualPosition);
    meiASSERT(returnValue == MPIMessageOK);

    meiPlatformSleep(15);
};
```

scopen.c -- Transition commutation from close-loop to open-loop mode

```
/* ScOpen.C */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Transition commutation from close-loop to open-loop mode

Warning! This Sample Program Sets DAC output level = 1.0 Volt

Check your system's safe continuous level (i.e., check amplifier
current gain and safe motor continuous current level)

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main ScOpen

argMainRENAME(main, ScOpen)
#endif

/* Command line arguments and defaults */
long    motorNumber = 0;

Arg argList[] = {
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};
```

```
int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle  */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIMotor    motor;  /* motor handle */

    MEIMotorConfig  motorConfigXmp;

    long    returnValue;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments  */
    if ((argIndex < argc) ||
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-motor # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
```

```
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    printf("Going to open loop ...");

    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Commutation.Mode = MEIXmpCommModeOPEN_LOOP;

    /* Warning!!!!! Setting DAC output level = 1.0 Volt */
    motorConfigXmp.Commutation.OutputLevel = (float)3277.0;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Delete objects */
    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

scstep.c -- Single Axis Sinusoidal Commutation Sample Configuration Program

```
/* ScStep.C  */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Single Axis Sinusoidal Commutation Sample Configuration Program

  Axis 0 -  Uses Stepper type initialization with a small
            90 (elec.) degree move.  Axis stays in open-loop mode
            with amplifier enabled.

  Note -    #def provided to reverse encoder and DAC phasing.
            Program disables HW Limits, Amp Fault, Home and
            Error Limit.

  Configure the XMP:

            Axis 0  Filter 0   Motor 0


 Note that the commutation configuration parameters will be different
 for your application.  Take particular care in setting the open-loop
 continuous DAC OutputLevel as this must be a safe level for your specific
 drive and motor combination.  Please read the MEI Sinusoidal Commutation
 Documentation.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main StepSCMain

argMainRENAME(main, StepSC)
#endif
```

```
#define AXIS_CNT    (1)

/* Command line arguments and defaults */
long    axisNumber      = 0;
long    motionNumber    = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* option to reverse encoder phasing */
#undef ReverseEncPhasing
/* Option to reverse DAC phasing */
#undef ReverseDACPhasing


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis handles */
    MPIFilter   filter;     /* filter handles */
    MPIMotor    motor;      /* motor handles  */

    MPIControlConfig    controlConfig;  /* motion controller configuration */
    MPIAxisConfig       axisConfig;     /* axis configuration */
    MPIFilterGain       gain;
    MPIMotorConfig      motorConfig;
    MEIMotorConfig      motorConfigXmp;
    MEIFilterGainPID    *pid;
    MEIXmpCommutationBlock  *commutation;

    MPIMotion       motion; /* Coordinated motion object handles */

    MPIControlType          controlType;
    MPIControlAddress       controlAddress;

    long    returnValue;    /* return value from library */

    long    nEncCountPerCycle = 4096; /* number of encoder counts per motor
revolution */
    float   OpenDAC_Level   = (float)3277.0;  /* Open Loop DAC Level where 32767. =
10V) */
    float   N_CYCLES    =       (float)3.0;     /* number of elec. cycles per motor
rev. */

#if defined(ReverseDACPhasing)
    long    dacCount;
    long    dacNumber[MPIMotorDacCountMAX];
    long    number;
#endif
```

```
    float   inPositionLimit = (float)1.0e10;

    long    argIndex, i;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber   >= MEIXmpMAX_Axes + AXIS_CNT) ||
        (motionNumber >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_CNT,
                            MEIXmpMAX_MSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create device driver motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /*
       Configure motion controller
       Set Sample Rate = 2 kHz
       Configure for commutation B phase DACs and ADCs
     */
    returnValue =
        mpiControlConfigGet(control,
```

```
                        &controlConfig,
                        NULL);
    msgCHECK(returnValue);

    controlConfig.sampleRate = 2000;

    /*
        When supporting commutation,
        Cmd and Aux DACs must be enabled.
        Enabling the same number of Cmd and Aux DACs as there are motors.
     */
    controlConfig.cmdDacCount = controlConfig.motorCount;
    controlConfig.auxDacCount = controlConfig.motorCount;

    /* Configure required number of A/D */
    controlConfig.adcCount = 8;

    returnValue =
        mpiControlConfigSet(control,
                        &controlConfig,
                        NULL);
    msgCHECK(returnValue);

    /* Create axis, filter and motor controller objects */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    filter =
        mpiFilterCreate(control,
                      axisNumber);
    msgCHECK(mpiFilterValidate(filter));

    motor =
        mpiMotorCreate(control,
                      axisNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Disable the amplifier */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                        FALSE);
    msgCHECK(returnValue);

    /*   Configure axis  */
    returnValue =
        mpiAxisConfigGet(axis,
                       &axisConfig,
                       NULL);
    msgCHECK(returnValue);

    /* Set inPositionLimit */
    axisConfig.inPosition.tolerance.positionFine = inPositionLimit;

    returnValue =
        mpiAxisConfigSet(axis,
```

```
                        &axisConfig,
                        NULL);
    msgCHECK(returnValue);

    /* Configure Motor */
    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /*
       Disable limit error, home, amp fault & H/W Limits
       Set encoder phase
     */

    /* Disable limit error */
    motorConfig.event[MPIEventTypeLIMIT_ERROR].action           = MPIActionNONE;
    motorConfig.event[MPIEventTypeLIMIT_ERROR].trigger.error    = (float)50000;

    /* Disable home event */
    motorConfig.event[MPIEventTypeHOME].action  = MPIActionNONE;

    /* Disable amp fault event */
    motorConfig.event[MPIEventTypeAMP_FAULT].action = MPIActionNONE;

    /* Disable neg. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_NEG].action  = MPIActionNONE;

    /* Disable pos. HW limit event */
    motorConfig.event[MPIEventTypeLIMIT_HW_POS].action  = MPIActionNONE;

    /*
       Set encoder phase:

       0 - for normal
       1 - for reversed
     */

#ifdef ReverseEncPhasing
    motorConfig.encoderPhase = 1;
#else
    motorConfig.encoderPhase = 0;
#endif

    /* Setup commutation  */
    commutation = &motorConfigXmp.Commutation;

    /* Motor Controller commutation parameters */
    commutation->Mode        = MEIXmpCommModeOPEN_LOOP;
    commutation->Length      = nEncCountPerCycle;
    commutation->Scale       = (float)MEIXmpCOMM_TABLE_SIZE * N_CYCLES /
                               (float)commutation->Length;
    commutation->PhaseDelta  = MEIXmpCOMM_120DEGREES;
    commutation->OutputLevel = (float)0.0;   /* level will be ramped after enable */
    commutation->Offset      = (long)0;
```

```
#if defined(ReverseDACPhasing)
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseINVERTED;
#else
    motorConfigXmp.Dac.Phase = MEIXmpDACPhaseNORMAL;
#endif

    /* Set configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Get current configuration */
    returnValue =
        mpiFilterGainGet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);

    pid = (MEIFilterGainPID *)&gain;

    /* Filter Controller control parameters */
    pid->gain.proportional          = (float)100.;
    pid->gain.integral              = (float)0.0;
    pid->gain.derivative            = (float)400.;
    pid->feedForward.velocity       = (float)0.0;
    pid->feedForward.acceleration   = (float)0.0;
    pid->feedForward.friction       = (float)0.0;
    pid->integrationMax.moving      = (float)0.0;
    pid->integrationMax.rest        = (float)0.0;
    pid->output.limit               = (float)32767.0;
    pid->output.offset              = (float)0.0;

    /* Set filter configuration  */
    returnValue =
        mpiFilterGainSet(filter,
                         MEIFilterGainIndexDEFAULT,
                         &gain);
    msgCHECK(returnValue);

    /*
       Create a Motion Object and append an Axis
       Note 1 - separate axis appends will be required for coordinated motion
       Note 2 - Axis Map does not take effect until motion is commanded
     */
    motion =
         mpiMotionCreate(control,
                         motionNumber,
                         axis);
    msgCHECK(mpiMotionValidate(motion));

    /*  Clear out any previous errors */
    returnValue =
            mpiMotionAction(motion,MPIActionRESET);
    msgCHECK(returnValue);
```

```
    /*  Immediately send Map to controller  */
    returnValue =
        mpiMotionAction(motion,
                        (MPIAction)MEIActionMAP);
    msgCHECK(returnValue);

    /*  Enable the amplifier */
    returnValue =
        mpiMotorAmpEnableSet(motor,
                             TRUE);
    msgCHECK(returnValue);

    /*
        DAC output will be ramped and stage will be pulled to a magnetic pole.
        Next, delay for stage to settle.  Implement outbound move,delay, then
        proceed with inbound move and delay.
     */

    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfig,
                          &motorConfigXmp);
    msgCHECK(returnValue);


    for(i= 1; i <= 100; i++){

        commutation->OutputLevel = (float)(i)*((OpenDAC_Level)/((float)100.0));

        /* Set configuration */
        returnValue =
            mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
        msgCHECK(returnValue);


        meiPlatformSleep(30);
    }

    meiPlatformSleep(1000);

    /* Motor outbound move */

    for(i= 1; i <= 100; i++){

        commutation->Offset      = (long)(i*2.56);

        /* Set configuration */
        returnValue =
            mpiMotorConfigSet(motor,
                          &motorConfig,
                          &motorConfigXmp);
        msgCHECK(returnValue);

        meiPlatformSleep(30);
```

```
    }

    meiPlatformSleep(1000);

    /* Motor return move */
    for(i= 1; i <= 100; i++){


        commutation->Offset      = (long)(256 - (i*2.56));

            /* Set configuration */
        returnValue =
            mpiMotorConfigSet(motor,
                        &motorConfig,
                        &motorConfigXmp);
        msgCHECK(returnValue);

        meiPlatformSleep(30);
    }

    meiPlatformSleep(1000);


    /* Delete Objects */
    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiFilterDelete(filter);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

scview.c -- Display commutation parameters

```
/* ScView.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */


/*

:Display commutation parameters

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    ScView

argMainRENAME(main, ScView)
#endif

/* Command line arguments and defaults */
long    motorNumber = 0;

Arg argList[] = {
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

char *modeLabel[] = {
    "MEIXmpCommModeNONE",
    "MEIXmpCommModeCLOSED_LOOP",
```

```
    "MEIXmpCommModeOPEN_LOOP",
    "MEIXmpCommModeSIMULATE",
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;        /* motion controller handle */
    MPIMotor        motor;          /* motor */
    MEIMotorConfig  motorConfig;    /* motor config structure */

    long    returnValue;    /* return value from library */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-motor # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
    }
```

```
    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create motor object using motorNumber */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfig);
    msgCHECK(returnValue);

    printf("Motor %ld:\n",
            motorNumber);

    printf("\tCommutation Mode = %ld (%s)\n",
            motorConfig.Commutation.Mode,
            modeLabel[motorConfig.Commutation.Mode]);

    printf("\tCommutation Length = %ld\n",
            motorConfig.Commutation.Length);

    printf("\tCommutation Scale = %f\n",
            motorConfig.Commutation.Scale);

    printf("\tCommutation Phase Delta = %ld\n",
            motorConfig.Commutation.PhaseDelta);

    printf("\tCommutation Theta = %ld\n",
            motorConfig.Commutation.Theta);

    if (motorConfig.Commutation.Length != 0) {

        double  phaseAngle;
        long tableIndex;

        tableIndex =
            ((long)(motorConfig.Commutation.Theta *
              motorConfig.Commutation.Scale)) &
             (MEIXmpCOMM_TABLE_SIZE - 1);

        phaseAngle =
```

```
                (double)tableIndex * 360.0/
                (double)MEIXmpCOMM_TABLE_SIZE;

            printf("\tPhase angle = %lf degrees\n",
                    phaseAngle);
        }

        /* Delete objects */
        returnValue = mpiMotorDelete(motor);
        msgCHECK(returnValue);

        returnValue = mpiControlDelete(control);
        msgCHECK(returnValue);

        return ((int)returnValue);
    }
```

seq1.c -- Perform a repeated multi-axis motion command sequence

```
/* seq1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seq1.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a repeated multi-axis motion command sequence

This sample program creates a program sequencer that will generate a multi-axis
 motion beween two points.  The program sequencer waits for the motion to
 complete in between moves.  After commanding the two motions the program
 sequencer loops back to the beginning of the motion command sequence, and
 repeats the motion until the user presses a key.  The program execution is
 handled entirely on the controller, with no host computer intervention.  The
 motion can be viewed by the Motion Console and Motion Scope utilities.

The XMP program sequencer controls the execution of a single command or a
 series of commands on the XMP controller.  The program sequencer provides the
 ability to execute programs directly on the XMP controller without host
 intervention.  Examples of individual commands that can be executed by the
 program sequencer are motion, looping, conditional branching, computation,
 reading and writing of memory, time delays, waiting for conditions, setting
 inputs/outputs, and generation of events.  This rich command set provides
 capability similar to, and beyond, that provided by Programmable Logic
 Controller (PLC) programs.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#if defined(ARG_MAIN_RENAME)
#define main    seq1Main

argMainRENAME(main, seq1)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
long            sequenceNumber  = 0;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",        ArgTypeLONG,    &axisNumber[0],     },
    {   "-motion",      ArgTypeLONG,    &motionNumber,      },
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,    },
    {   "-type",        ArgTypeLONG,    &motionType,        },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 200000.0, 0.0,        },
    { 0.0,      200000.0,   },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
};

/* Motion Parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
```

```
    {   &trajectory[1][0],  },
};

/* 2 => START + MOTION_DONE 1 => BRANCH */
MPICommand  CommandTable[(MOTION_COUNT * 2) + 1];
#define COMMAND_COUNT   (sizeof(CommandTable) / sizeof(MPICommand))

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIMotion   motion;             /* motion handle */
    MPIAxis     axis[AXIS_COUNT];   /* axis handles */
    MPISequence sequence;           /* sequence handle */

    MPICommandParams    commandParams;  /* command parameters */

    long    returnValue;    /* return value from library */

    long    commandIndex;   /* CommandTable[] index */
    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]  > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber   >= MEIXmpMAX_MSs) ||
        (sequenceNumber >= MEIXmpMAX_PSs) ||
```

```
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_PSs - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create motion object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create axis object for axis #0 */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);   /* axis #0 */
```

```
    msgCHECK(mpiAxisValidate(axis[0]));

    /* Create axis object for axis #1 */
    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);   /* axis #1 */
    msgCHECK(mpiAxisValidate(axis[1]));

    /* Create motion axis list */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axis);
    msgCHECK(returnValue);

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          -1);
    msgCHECK(mpiSequenceValidate(sequence));

    /* CommandTable[commandIndex] */
    commandIndex = 0;

    /* Create motion Commands */
    for (index = 0; index < MOTION_COUNT; index++) {
        MPIMotionParams *motionParams;

        /* mpiMotionStart(motion, type, params); */
        commandParams.motion.motionCommand  = MPICommandMotionSTART;
        commandParams.motion.motion         = motion;
        commandParams.motion.type           = motionType;

        motionParams = &commandParams.motion.params;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams->sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams->trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams->velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }
```

```
        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeMOTION,
                             &commandParams,
                             (commandIndex == 0) ? "First" : NULL);

        returnValue = mpiCommandValidate(CommandTable[commandIndex]);
        msgCHECK(returnValue);

        commandIndex++;

        commandParams.waitEvent.handle  = motion;
        commandParams.waitEvent.oper    = MPICommandOperatorBIT_SET;
        mpiEventMaskCLEAR(commandParams.waitEvent.mask);
        mpiEventMaskSET(commandParams.waitEvent.mask, MPIEventTypeMOTION_DONE);

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeWAIT_EVENT,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;
    }

    /* Branch to the first command of the sequence */
    commandParams.branch.label      = "First";  /* First command */
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* Create sequence command list */
    returnValue =
        mpiSequenceCommandListSet(sequence,
                                  commandIndex,
                                  CommandTable);
    msgCHECK(returnValue);

    /* Start sequence */
    returnValue =
        mpiSequenceStart(sequence,
                         MPIHandleVOID);
    msgCHECK(returnValue);

    meiPlatformConsole("Press any key to stop sequence\n");
```

```
    meiPlatformKey(MPIWaitFOREVER);

    /* Stop sequence */
    returnValue = mpiSequenceStop(sequence);
    msgCHECK(returnValue);

    /* Delete Objects */
    returnValue = mpiSequenceDelete(sequence);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (index = 0; index < AXIS_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

seq2.c -- Perform a repeated multi-axis motion command sequence, wait for an I/O bit

```
/* seq2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seq2.c 13    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a repeated multi-axis motion command sequence, wait for an I/O bit

This sample program creates a program sequencer that will generate a multi-axis
 motion beween two points.  The program sequencer waits for the motion to
 complete in between moves.  After commanding the two motions the program
 sequencer waits for a transceiver I/O bit to change state (defined by
 TRANSCEIVER_ID and TRIGGER_STATE at the top of the program).  Then the sequence
 loops back to the beginning of the motion command sequence, and repeats the
 motion until the user presses a key.  The program execution is handled entirely
 on the controller, with no host computer intervention.  The motion can be
 viewed by the Motion Console and Motion Scope utilities.

The XMP program sequencer controls the execution of a single command or a
 series of commands on the XMP controller.  The program sequencer provides the
 ability to execute programs directly on the XMP controller without host
 intervention.  Examples of individual commands that can be executed by the
 program sequencer are motion, looping, conditional branching, computation,
 reading and writing of memory, time delays, waiting for conditions, setting
 inputs/outputs, and generation of events.  This rich command set provides
 capability similar to, and beyond, that provided by Programmable Logic
 Controller (PLC) programs.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"
```

```
#if defined(ARG_MAIN_RENAME)
#define main    seq2Main

argMainRENAME(main, seq2)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Define the transceiver to wait for in between motion sequences */
#define TRANSCEIVER_ID  (MEIMotorTransceiverIdA)  /* A, B, or C */

/* Define the trigger value for the transceiver,
   MPICommandOperatorBIT_SET or MPICommandOperatorBIT_CLEAR
*/
#define TRIGGER_STATE   (MPICommandOperatorBIT_SET)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
long            motorNumber     = 0;
long            sequenceNumber  = 0;
long            sequenceSize    = -1;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",        ArgTypeLONG,    &axisNumber[0],     },
    {   "-motion",      ArgTypeLONG,    &motionNumber,      },
    {   "-motor",       ArgTypeLONG,    &motorNumber,       },
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,    },
    {   "-size",        ArgTypeLONG,    &sequenceSize,      },
    {   "-type",        ArgTypeLONG,    &motionType,        },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  0.0,        },
    { 0.0,      20000.0,    },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
};

/* motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
```

```
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};


                        /* 3 => START + MOTION_DONE + I/O 1 => BRANCH */
MPICommand  CommandTable[(MOTION_COUNT * 3) + 1];
#define COMMAND_COUNT   (sizeof(CommandTable) / sizeof(MPICommand))

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIMotion   motion;             /* motion handle */
    MPIAxis     axis[AXIS_COUNT];   /* axis handles */
    MPIMotor    motor;              /* motor handle */

    MPISequence sequence;           /* sequence handle */

    MPICommandParams    commandParams;  /* command parameters */

    long    returnValue;    /* return value from library */

    long    commandIndex;   /* CommandTable[] index */
    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
```

```
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]  > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber   >= MEIXmpMAX_MSs) ||
        (motorNumber    >= MEIXmpMAX_Motors) ||
        (sequenceNumber >= MEIXmpMAX_PSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[-size # (%d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1,
                            MEIXmpMAX_PSs - 1,
                            sequenceSize,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);
```

```
    /* Create motion object for MS number */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create axis object for X_AXIS */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis[0]));

    /* Create axis object for Y_AXIS */
    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axis[1]));

    /* Create motion axis list */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axis);
    msgCHECK(returnValue);

    /* Create motor object */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          sequenceSize);
    msgCHECK(mpiSequenceValidate(sequence));

    /* CommandTable[commandIndex] */
    commandIndex = 0;

    /* Create motion Commands */
    for (index = 0; index < MOTION_COUNT; index++) {
        MPIMotionParams *motionParams;

        /* mpiMotionStart(motion, type, params); */
        commandParams.motion.motionCommand  = MPICommandMotionSTART;
        commandParams.motion.motion         = motion;
        commandParams.motion.type           = motionType;

        motionParams = &commandParams.motion.params;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams->sCurve = sCurve[index];
```

```
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams->trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams->velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeMOTION,
                             &commandParams,
                             (commandIndex == 0) ? "First" : NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;

        commandParams.waitEvent.handle  = motion;
        commandParams.waitEvent.oper    = MPICommandOperatorBIT_SET;
        mpiEventMaskCLEAR(commandParams.waitEvent.mask);
        mpiEventMaskSET(commandParams.waitEvent.mask, MPIEventTypeMOTION_DONE);

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeWAIT_EVENT,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;

        /* while (mpiControlUserIoGet(control, &io), (io.input[0] & mask) == 0); */
        commandParams.waitIO.type           = MPIIoTypeMOTOR;
        commandParams.waitIO.source.motor   = motor;
        commandParams.waitIO.oper           = TRIGGER_STATE;
        commandParams.waitIO.mask           = 0x1 << TRANSCEIVER_ID;    /* bit 0 */

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeWAIT_IO,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;
    }

    /* Branch to the first command of the sequence */
    commandParams.branch.label      = "First";  /* First command */
```

```
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* Create sequence command list */
    returnValue =
        mpiSequenceCommandListSet(sequence,
                                  commandIndex,
                                  CommandTable);
    msgCHECK(returnValue);

    /* Start sequence */
    returnValue =
        mpiSequenceStart(sequence,
                         MPIHandleVOID);

    if (returnValue != MPIMessageOK) {
        fprintf(stderr, "%s: mpiSequenceStart() returns 0x%x: %s\n",
                        argv[0],
                        returnValue,
                        mpiMessage(returnValue, NULL));
        exit(2);
    }

    meiPlatformConsole("Press any key to stop sequence\n");
    meiPlatformKey(MPIWaitFOREVER);

    /* Stop sequence */
    returnValue = mpiSequenceStop(sequence);
    msgCHECK(returnValue);

    returnValue = mpiSequenceDelete(sequence);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    for (index = 0; index < AXIS_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

seq3.c -- Wait for an I/O bit to Toggle Before Commanding Motion with a Sequencer

```
/* seq3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seq3.c 12    7/23/01 2:36p Kevinh $";
#endif

/*

:Wait for an I/O bit to Toggle Before Commanding Motion with a Sequencer

Wait for an I/O bit between moves.
This program will wait for motor 0's Transceiver A to go high.
Use an event manager to handle flow control.


Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    seq3Main

argMainRENAME(main, seq3)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
long            motorNumber     = 0;
long            sequenceNumber  = 0;
```

```
long            sequenceSize    = -1;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",        ArgTypeLONG,    &axisNumber[0],     },
    {   "-motion",      ArgTypeLONG,    &motionNumber,      },
    {   "-motor",       ArgTypeLONG,    &motorNumber,       },
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,    },
    {   "-size",        ArgTypeLONG,    &sequenceSize,      },
    {   "-type",        ArgTypeLONG,    &motionType,        },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  0.0,        },
    { 0.0,      20000.0,    },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
};

/* motion parameters */

MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

                        /* 4 => START + MOTION_DONE + I/O + EVENT 1 => BRANCH */
MPICommand  CommandTable[(MOTION_COUNT * 4) + 1];
#define COMMAND_COUNT   (sizeof(CommandTable) / sizeof(MPICommand))

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIMotion   motion;             /* motion handle */
```

```
    MPIAxis     axis[AXIS_COUNT];   /* axis handles */
    MPIMotor    motor;              /* motor handle */

    MPISequence sequence;           /* sequence handle */
    MPIEventMgr eventMgr;           /* event manager handle */

    Service service;

    MPICommandParams    commandParams;  /* command parameters */

    long    returnValue;    /* return value from library */

    long    commandIndex;   /* CommandTable[] index */

    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]  > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber   >= MEIXmpMAX_MSs) ||
        (motorNumber    >= MEIXmpMAX_Motors) ||
        (sequenceNumber >= MEIXmpMAX_PSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[-size # (%d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
```

```
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - AXIS_COUNT,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1,
                            MEIXmpMAX_PSs - 1,
                            sequenceSize,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create motion object for MS number */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create axis object for X_AXIS number */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis[0]));

    /* Create axis object for Y_AXIS number */
    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axis[1]));
```

```
    /* Create motion axis list */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axis);
    msgCHECK(returnValue);

    /* Create motor objects */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          sequenceSize);
    msgCHECK(mpiSequenceValidate(sequence));

#if defined(_DEBUG)
    returnValue =
        meiObjectTraceSet(sequence,
                          MEISequenceTraceLOAD);
    msgCHECK(returnValue);
#endif

    /* CommandTable[commandIndex] */
    commandIndex = 0;

    /* Create motion Commands */
    for (index = 0; index < MOTION_COUNT; index++) {
        MPIMotionParams *motionParams;

        /* mpiMotionStart(motion, type, params); */
        commandParams.motion.motionCommand  = MPICommandMotionSTART;
        commandParams.motion.motion         = motion;
        commandParams.motion.type           = motionType;

        motionParams = &commandParams.motion.params;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams->sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams->trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams->velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
```

```
    }
}

/* Create Command */
CommandTable[commandIndex] =
    mpiCommandCreate(MPICommandTypeMOTION,
                     &commandParams,
                     (commandIndex == 0) ? "First" : NULL);
msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

commandIndex++;

commandParams.waitEvent.handle  = motion;
commandParams.waitEvent.oper    = MPICommandOperatorBIT_SET;
mpiEventMaskCLEAR(commandParams.waitEvent.mask);
mpiEventMaskSET(commandParams.waitEvent.mask, MPIEventTypeMOTION_DONE);

/* Create Command */
CommandTable[commandIndex] =
    mpiCommandCreate(MPICommandTypeWAIT_EVENT,
                     &commandParams,
                     NULL);
msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

commandIndex++;

commandParams.branchEvent.label     = NULL;
commandParams.branchEvent.handle    = motion;
commandParams.branchEvent.oper      = MPICommandOperatorBIT_SET;
mpiEventMaskCLEAR(commandParams.branchEvent.mask);
mpiEventMaskMOTOR(commandParams.branchEvent.mask);

/* Create Command */
CommandTable[commandIndex] =
    mpiCommandCreate(MPICommandTypeBRANCH_EVENT,
                     &commandParams,
                     NULL);

msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

commandIndex++;

/* while (mpiControlUserIoGet(control, &io), (io.input[0] & mask) == 0); */
commandParams.waitIO.type           = MPIIoTypeMOTOR;
commandParams.waitIO.source.motor   = motor;

commandParams.waitIO.oper           = MPICommandOperatorBIT_SET;
commandParams.waitIO.mask           = 0x1 << 0; /* bit 0 */

/* Create Command */
CommandTable[commandIndex] =
    mpiCommandCreate(MPICommandTypeWAIT_IO,
                     &commandParams,
                     NULL);
msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

commandIndex++;
```

```
    }

    /* Branch to the first command of the sequence */
    commandParams.branch.label      = "First";  /* First command */
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* Create sequence command list */
    returnValue =
        mpiSequenceCommandListSet(sequence,
                                  commandIndex,
                                  CommandTable);
    msgCHECK(returnValue);

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Start sequence */
    returnValue =
        mpiSequenceStart(sequence,
                         MPIHandleVOID);

    if (returnValue != MPIMessageOK) {
        fprintf(stderr, "%s: mpiSequenceStart() returns 0x%x: %s\n",
                        argv[0],
                        returnValue,
                        mpiMessage(returnValue, NULL));
        exit(2);
    }

    meiPlatformConsole("Press any key to stop sequence\n");
    meiPlatformKey(MPIWaitFOREVER);

    /* Stop sequence */
    returnValue = mpiSequenceStop(sequence);
    msgCHECK(returnValue);

    returnValue = mpiSequenceDelete(sequence);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
```

```
    msgCHECK(returnValue);

    for (index = 0; index < AXIS_COUNT; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

seq4.c -- Perform a repeated multi-axis motion sequence, wait for I/O, & monitor location

```
/* seq4.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seq4.c 20    7/23/01 2:36p Kevinh $";
#endif

/*

:Perform a repeated multi-axis motion sequence, wait for I/O, & monitor location



This sample program creates a program sequencer that will generate a multi-axis
 motion beween two points.  The program sequencer waits for the motion to
 complete in between moves.  After commanding the two motions the program
 sequencer waits for a transceiver I/O bit to change state (defined by
 TRANSCEIVER_WAIT_ID and TRIGGER_STATE at the top of the program).  Then the
 sequence loops back to the beginning of the motion command sequence, and
 repeats the motion until the user presses a key.

In addition there is a second program sequencer that monitors the position of
 the X and Y axes.  If their positions fall within a rectangle, defined by the
 position array, then a transceiver bit is set (defined by TRANSCEIVER_SET_ID).
 This can be used to signal process or inspection equipment to begin their
 actions.

The program execution is handled entirely on the controller, with no host
 computer intervention.  The motion can be viewed by the Motion Console and
 Motion Scope utilities.

The XMP program sequencer controls the execution of a single command or a
 series of commands on the XMP controller.  The program sequencer provides the
 ability to execute programs directly on the XMP controller without host
 intervention.  Examples of individual commands that can be executed by the
 program sequencer are motion, looping, conditional branching, computation,
 reading and writing of memory, time delays, waiting for conditions, setting
 inputs/outputs, and generation of events.  This rich command set provides
 capability similar to, and beyond, that provided by Programmable Logic
 Controller (PLC) programs.

 Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
```

```
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    seq4Main

argMainRENAME(main, seq4)
#endif

#define MOTION_COUNT    (2)
#define AXIS_COUNT      (2)

/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0, 1, };
long            motionNumber    = 0;
long            motorNumber     = 0;
long            sequenceNumber  = 0;
long            sequenceSize    = 128;
MPIMotionType   motionType      = MPIMotionTypeTRAPEZOIDAL;

Arg argList[] = {
    {   "-axis",        ArgTypeLONG,    &axisNumber[0],     },
    {   "-motion",      ArgTypeLONG,    &motionNumber,      },
    {   "-motor",       ArgTypeLONG,    &motorNumber,       },
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,    },
    {   "-size",        ArgTypeLONG,    &sequenceSize,      },
    {   "-type",        ArgTypeLONG,    &motionType,        },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT][AXIS_COUNT] = {
    { 20000.0,  0.0,        },
    { 0.0,      20000.0,    },
};

MPITrajectory trajectory[MOTION_COUNT][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      100000.0,   100000.0,   50.0,   },
        { 10000.0,      100000.0,   100000.0,   50.0,   },
    },
};

/* motion parameters */
```

```
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

typedef enum {
    ExternalEventSEQUENCE_DONE,
} ExternalEvent;

                        /* 4 => START + MOTION_DONE + I/O + EVENT */
                        /* 2 => EXTERNAL_EVENT + BRANCH */
MPICommand  CommandTable[(MOTION_COUNT * 4) + 2];
#define COMMAND_COUNT   (sizeof(CommandTable) / sizeof(MPICommand))


#define IO_CONFIG   (MEIMotorTransceiverConfigOUTPUT) /* INPUT or OUTPUT  */
#define INVERT_BIT  (FALSE)


/* Toggle the transceiver associated with the motor.  In this case, toggle
   transceiver B
*/
#define TRANSCEIVER_SET_ID  (MEIMotorTransceiverIdB)  /* A, B, or C, */
/* Define the transceiver to wait for in between motion sequences */
#define TRANSCEIVER_WAIT_ID (MEIMotorTransceiverIdA)  /* A, B, or C */

/* Define the trigger value for the transceiver,
   MPICommandOperatorBIT_SET or MPICommandOperatorBIT_CLEAR
*/
#define TRIGGER_STATE   (MPICommandOperatorBIT_SET)

MPISequence
    monitorPosition(MPIControl  control,
                    MPIMotor    motor,
                    MPIAxis     axisX,
                    MPIAxis     axisY,
                    double      *limitX,
                    double      *limitY);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;            /* motion controller handle */
    MPIMotion   motion;             /* motion handle */
    MPIAxis     axis[AXIS_COUNT];   /* axis handles */
    MPIMotor    motor;              /* motor handle */
```

```
    MPISequence sequence;           /* sequence handle */
    MPISequence sequenceMonitor;    /* sequence handle */
    MPINotify   notify;             /* notification handle */
    MPIEventMgr eventMgr;           /* event manager handle */

    MPIEventMask    eventMask;

    Service service;

    MPICommandType      type;           /* command type */
    MPICommandParams    commandParams;  /* command parameters */

    double  limitX[AXIS_COUNT];
    double  limitY[AXIS_COUNT];
    double  limit;

    long    returnValue;    /* return value from library */

    long    commandIndex;   /* CommandTable[] index */
    long    index;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0]  > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber   >= MEIXmpMAX_MSs) ||
        (motorNumber    >= MEIXmpMAX_Motors) ||
        (sequenceNumber >= MEIXmpMAX_PSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
```

```
                            "\t\t[-axis # (0 .. %d)]\n"
                            "\t\t[-motion # (0 .. %d)]\n"
                            "\t\t[-motor # (0 .. %d)]\n"
                            "\t\t[-sequence # (0 .. %d)]\n"
                            "\t\t[-size # (%d)]\n"
                            "\t\t[-type # (0 .. %d)]\n",
                             argv[0],
                             ArgUSAGE,
                             MEIXmpMAX_Axes - AXIS_COUNT,
                             MEIXmpMAX_MSs - 1,
                             MEIXmpMAX_Motors - 1,
                             MEIXmpMAX_PSs - 1,
                             sequenceSize,
                             MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    axisNumber[1] = axisNumber[0] + 1;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create motion object for MS number */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        MPIHandleVOID);
    msgCHECK(mpiMotionValidate(motion));

    /* Create axis object for X_AXIS number */
    axis[0] =
        mpiAxisCreate(control,
                      axisNumber[0]);
    msgCHECK(mpiAxisValidate(axis[0]));
```

```
    /* Create axis object for Y_AXIS number */
    axis[1] =
        mpiAxisCreate(control,
                      axisNumber[1]);
    msgCHECK(mpiAxisValidate(axis[1]));

    /* Create motion axis list */
    returnValue =
        mpiMotionAxisListSet(motion,
                             AXIS_COUNT,
                             axis);
    msgCHECK(returnValue);

    /* Request notification of all events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create motor objects */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          sequenceSize);
    msgCHECK(mpiSequenceValidate(sequence));

#if defined(_DEBUG)
    returnValue =
        meiObjectTraceSet(sequence,
                          MEISequenceTraceLOAD);
    msgCHECK(returnValue);
#endif

    /* Request notification of all events from sequence */
    returnValue =
        mpiSequenceEventNotifySet(sequence,
                                  eventMask,
                                  NULL);
    msgCHECK(returnValue);

    /* CommandTable[commandIndex] */
    commandIndex = 0;

    /* Create motion Commands */
    for (index = 0; index < MOTION_COUNT; index++) {
        MPIMotionParams *motionParams;

        /* mpiMotionStart(motion, type, params); */
```

```
        type = MPICommandTypeMOTION;

        commandParams.motion.motionCommand  = MPICommandMotionSTART;
        commandParams.motion.motion         = motion;
        commandParams.motion.type           = motionType;

        motionParams = &commandParams.motion.params;

        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams->sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams->trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams->velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(type,
                             &commandParams,
                             (commandIndex == 0) ? "First" : NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;

        commandParams.waitEvent.handle  = motion;
        commandParams.waitEvent.oper    = MPICommandOperatorBIT_SET;
        mpiEventMaskCLEAR(commandParams.waitEvent.mask);
        mpiEventMaskSET(commandParams.waitEvent.mask, MPIEventTypeMOTION_DONE);

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeWAIT_EVENT,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;

        commandParams.branchEvent.label     = NULL;
        commandParams.branchEvent.handle    = motion;
        commandParams.branchEvent.oper      = MPICommandOperatorBIT_SET;
        mpiEventMaskCLEAR(commandParams.branchEvent.mask);
        mpiEventMaskMOTOR(commandParams.branchEvent.mask);

        /* Create Command */
```

```
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeBRANCH_EVENT,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;

        /* while (mpiControlUserIoGet(control, &io), (io.input[0] & mask) == 0); */
        commandParams.waitIO.type           = MPIIoTypeMOTOR;
        commandParams.waitIO.source.motor   = motor;
        commandParams.waitIO.oper           = TRIGGER_STATE;
        commandParams.waitIO.mask           = 0x1 << TRANSCEIVER_WAIT_ID;   /* bit 0
 */

        /* Create Command */
        CommandTable[commandIndex] =
            mpiCommandCreate(MPICommandTypeWAIT_IO,
                             &commandParams,
                             NULL);
        msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

        commandIndex++;
    }

    /* MPIEventTypeEXTERNAL */
    commandParams.event.value = ExternalEventSEQUENCE_DONE;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeEVENT,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* Branch to the first command of the sequence */
    commandParams.branch.label      = "First";  /* First command */
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    /* Create Command */
    CommandTable[commandIndex] =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(CommandTable[commandIndex]));

    commandIndex++;

    /* Create sequence command list */
    returnValue =
        mpiSequenceCommandListSet(sequence,
                                  commandIndex,
                                  CommandTable);
    msgCHECK(returnValue);
```

```
    /* Create event notification object */
    notify =
        mpiNotifyCreate(eventMask,      /* Notify of all MPI events */
                        MPIHandleVOID); /* Notify from all sources */
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    limit =
        fabs(position[0][0] - position[1][0]) /
        (AXIS_COUNT + 2);

    if (position[0][0] < position[1][0]) {
        limitX[0] = position[0][0] + limit;
        limitX[1] = position[1][0] - limit;
    }
    else {
        limitX[0] = position[1][0] + limit;
        limitX[1] = position[0][0] - limit;
    }

    limit =
        fabs(position[0][1] - position[1][1]) /
        (AXIS_COUNT + 2);

    if (position[0][1] < position[1][1]) {
        limitY[0] = position[0][1] + limit;
        limitY[1] = position[1][1] - limit;
    }
    else {
        limitY[0] = position[1][1] + limit;
        limitY[1] = position[0][1] - limit;
    }

    sequenceMonitor =
        monitorPosition(control,
                        motor,
                        axis[0],
                        axis[1],
                        limitX,
                        limitY);
    msgCHECK(mpiSequenceValidate(sequenceMonitor));
```

```
    /* Start sequence */
    returnValue =
        mpiSequenceStart(sequence,
                         MPIHandleVOID);

    if (returnValue != MPIMessageOK) {
        fprintf(stderr, "%s: mpiSequenceStart() returns 0x%x: %s\n",
                        argv[0],
                        returnValue,
                        mpiMessage(returnValue, NULL));
        exit(2);
    }

    meiPlatformConsole("Press any key to stop sequence\n");

    while (returnValue == MPIMessageOK) {
        MPIEventStatus  status;

        returnValue =
            mpiNotifyEventWait(notify,
                               &status,
                               MPIWaitFOREVER);

        if (returnValue == MPIMessageOK) {
            MEIEventStatusInfo  *info;

            info = (MEIEventStatusInfo *)status.info;

            switch (status.type) {
                case MPIEventTypeNONE: {    /* ignore */
                    meiPlatformConsole("No event ...\n");
                    break;
                }
                case MPIEventTypeAMP_FAULT:
                case MPIEventTypeHOME:
                case MPIEventTypeLIMIT_ERROR:
                case MPIEventTypeLIMIT_HW_NEG:
                case MPIEventTypeLIMIT_HW_POS:
                case MPIEventTypeLIMIT_SW_NEG:
                case MPIEventTypeLIMIT_SW_POS: {    /* error */
                    long    number;

                    number = info->type.number;

                    meiPlatformConsole("Event %d on motor #%d\n",
                                       status.type,
                                       number);

                    break;
                }
                case MPIEventTypeMOTION_DONE: {
                    meiPlatformConsole("Motion done\n");
                    break;
                }
                case MPIEventTypeEXTERNAL: {
                    switch (info->type.value) {
```

```
                    case ExternalEventSEQUENCE_DONE: {
                        meiPlatformConsole("Sequence done\n");
                        break;
                    }
                    default: {
                        meiPlatformConsole("Unknown external event: %d\n",
                                           info->type.value);
                        break;
                    }
                }

                break;
            }
            default: {
                break;
            }
        }
    }

    if (returnValue == MPIMessageOK) {
        if (meiPlatformKey(MPIWaitPOLL) >= 0) {
            break;
        }
    }
}

meiPlatformConsole("Exiting ... returnValue 0x%x: %s\n",
                   returnValue,
                   mpiMessage(returnValue, NULL));

/* Stop sequence */
returnValue = mpiSequenceStop(sequenceMonitor);
msgCHECK(returnValue);

returnValue = mpiSequenceStop(sequence);
msgCHECK(returnValue);

returnValue = mpiSequenceDelete(sequenceMonitor);
msgCHECK(returnValue);

returnValue = mpiSequenceDelete(sequence);
msgCHECK(returnValue);

returnValue = mpiMotionDelete(motion);
msgCHECK(returnValue);

for (index = 0; index < AXIS_COUNT; index++) {
    returnValue = mpiAxisDelete(axis[index]);
    msgCHECK(returnValue);
}

returnValue = mpiMotorDelete(motor);
msgCHECK(returnValue);

returnValue = serviceDelete(service);
msgCHECK(returnValue);
```

```
    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

MPISequence
    monitorPosition(MPIControl  control,
                    MPIMotor    motor,
                    MPIAxis     axisX,
                    MPIAxis     axisY,
                    double      *limitX,
                    double      *limitY)
{
    MPISequence         sequence;
    MPICommand          command;
    MEIMotorConfig      motorConfigXmp;  /* contains transceiver configuration */
    MPICommandParams    commandParams;

    long        numberX;
    long        numberY;
    MEIXmpAxis  *xmpAxisX;
    MEIXmpAxis  *xmpAxisY;
    long        *positionX;
    long        *positionY;

    long    returnValue;

    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    meiASSERT(returnValue == MPIMessageOK);

    motorConfigXmp.Transceiver[TRANSCEIVER_SET_ID].Config = IO_CONFIG;
    motorConfigXmp.Transceiver[TRANSCEIVER_SET_ID].Invert = INVERT_BIT;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    meiASSERT(returnValue == MPIMessageOK);

    returnValue =
        mpiAxisNumber(axisX,
                      &numberX);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisNumber(axisY,
                      &numberY);
```

```
    msgCHECK(returnValue);

    returnValue =
        mpiAxisMemory(axisX,
                      (void **)&xmpAxisX);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisMemory(axisY,
                      (void **)&xmpAxisY);
    msgCHECK(returnValue);

    positionX = &xmpAxisX->ActualPosition;
    positionY = &xmpAxisY->ActualPosition;

    sequence =
        mpiSequenceCreate(control,
                          -1,
                          8);
    msgCHECK(mpiSequenceValidate(sequence));

    commandParams.branch.label              = "IoBitClear";
    commandParams.branch.expr.address.l     = positionX;
    commandParams.branch.expr.oper          = MPICommandOperatorLESS;
    commandParams.branch.expr.by.value.l    = (long)limitX[0];

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         "First");
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.branch.label              = "IoBitClear";
    commandParams.branch.expr.address.l     = positionX;
    commandParams.branch.expr.oper          = MPICommandOperatorGREATER;
    commandParams.branch.expr.by.value.l    = (long)limitX[1];

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.branch.label              = "IoBitClear";
    commandParams.branch.expr.address.l     = positionY;
    commandParams.branch.expr.oper          = MPICommandOperatorLESS;
    commandParams.branch.expr.by.value.l    = (long)limitY[0];
```

```
    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.branch.label              = "IoBitClear";
    commandParams.branch.expr.address.l     = positionY;
    commandParams.branch.expr.oper          = MPICommandOperatorGREATER;
    commandParams.branch.expr.by.value.l    = (long)limitY[1];

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.computeIO.type            = MPIIoTypeMOTOR;
    commandParams.computeIO.source.motor    = motor;
    commandParams.computeIO.oper            = MPICommandOperatorOR;
    commandParams.computeIO.mask            = 0x1 << TRANSCEIVER_SET_ID;    /* Set
bit 1 */

    command =
        mpiCommandCreate(MPICommandTypeCOMPUTE_IO,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.branch.label      = "First";
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
```

```
    msgCHECK(returnValue);

    commandParams.computeIO.type            = MPIIoTypeMOTOR;
    commandParams.computeIO.source.motor    = motor;
    commandParams.computeIO.oper            = MPICommandOperatorAND;
    commandParams.computeIO.mask            = ~(0x1 << TRANSCEIVER_SET_ID); /* Clear
bit 1 */

    command =
        mpiCommandCreate(MPICommandTypeCOMPUTE_IO,
                         &commandParams,
                         "IoBitClear");
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    commandParams.branch.label      = "First";
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &commandParams,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    returnValue =
        mpiSequenceStart(sequence,
                         MPIHandleVOID);
    msgCHECK(returnValue);

    return (sequence);
}
```

seqKill.c -- Program Sequence to monitor an I/O bit and abort all axes when active.

```
/* seqkill.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seqkill.c 9     7/18/01 9:32a Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    seqkillMain

argMainRENAME(main, seqkill)
#endif

/*

:Program Sequence to monitor an I/O bit and abort all axes when active.

This sample program creates a program sequencer that will monitor an I/O bit.
 When this I/O bit is set, it will abort all the axes from 0 to axisCount.  It
 will monitor motor 0's transceiver A to determine when to abort the axes.

The XMP program sequencer controls the execution of a single command or a
 series of commands on the XMP controller.  The program sequencer provides the
 ability to execute programs directly on the XMP controller without host
 intervention.  Examples of individual commands that can be executed by the
 program sequencer are motion, looping, conditional branching, computation,
 reading and writing of memory, time delays, waiting for conditions, setting
 inputs/outputs, and generation of events.  This rich command set provides
 capability similar to, and beyond, that provided by Programmable Logic
 Controller (PLC) programs.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"
```

```
#include "apputil.h"


#define MOTION_NUMBER   (0)
#define MOTOR_NUMBER    (0)
#define AXIS_COUNT      (8)

#define IO_TRIGGER      (MEIXmpMotorIOMaskXCVR_A)

/*
   Active High: MPICommandOperatorBIT_SET
   Active Low : MPICommandOperatorBIT_CLEAR
*/
#define IO_LEVEL        (MPICommandOperatorBIT_SET)


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisCount,
                 MPIMotor           *motor,
                 long                motorNumber)
{
    long            index;
    long            returnValue;


    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
```

```
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis objects */
    for (index = 0; index < axisCount; index++) {
        axis[index] =
            mpiAxisCreate(*control,
                          index);
        msgCHECK(mpiAxisValidate(axis[index]));
    }

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        *axis);
    msgCHECK(mpiMotionValidate(*motion));

    /* Create motor object */
    *motor =
        mpiMotorCreate(*control,
                        motorNumber);
    msgCHECK(mpiMotorValidate(*motor));

    /* Write Motion Supervisor AxisMap[], and clear fault conditions */
    returnValue =
        mpiMotionAction(*motion,
                        MPIActionRESET);
    msgCHECK(returnValue);
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis,
                    long             axisCount,
                    MPIMotor        *motor)
{
    long    index;
    long    returnValue;


    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);

    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
```

```
    msgCHECK(returnValue);

    /* Delete axis objects */
    for (index = 0; index < axisCount; index++) {
        returnValue = mpiAxisDelete(axis[index]);
        msgCHECK(returnValue);
    }

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/*
   sequenceCommandAdd() creates a command (MPICommand object) and appends it to
   sequence's list of commands.  This function does return MPI error codes so
   that sequence setup code may be more easily debugged.

   Warning:  sequenceCommandAdd() does not keep track of created command objects
   so it is important that before a sequence object is deleted, that each
   command object on sequence's list is itself deleted.  Otherwise, there will
   be memory leaks.  One can use sequenceProgramDelete() to accomplish this.
*/
long sequenceCommandAdd(MPISequence          sequence,
                        MPICommandType       commandType,
                        MPICommandParams    *commandParams,
                        const char          *label)
{
    MPICommand  command;
    long        returnValue;


    command =
        mpiCommandCreate(commandType,
                         commandParams,
                         label);
    returnValue =
        mpiCommandValidate(command);

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiSequenceCommandAppend(sequence,
                                     command);
    }

    return returnValue;
}


/*
   sequenceKillProgramCreate() creates a program that waits for one of motor's
   io lines (represented by trigger) to turn on then commands an E_STOP_ABORT.
*/
```

```
MPISequence sequenceKillProgramCreate(MPIMotion            motion,
                                      MPIMotor             motor,
                                      MEIXmpMotorIOMask    trigger,
                                      MPICommandOperator   level)
{
    MPISequence         sequence;
    MPICommandParams    commandParams;
    long                returnValue;


    /* Create sequence object */
    sequence =
        mpiSequenceCreate(mpiMotionControl(motion),
                          -1,   /* Use the next available sequence */
                          3);   /* # of commands in sequence */
    msgCHECK(mpiSequenceValidate(sequence));

    /* Wait for motor 0's bit 0 (transceiver A) to be set */
    commandParams.waitIO.type           = MPIIoTypeMOTOR;
    commandParams.waitIO.source.motor   = motor;
    commandParams.waitIO.oper           = level;
    commandParams.waitIO.mask           = trigger;

    returnValue =
        sequenceCommandAdd(sequence,
                           MPICommandTypeWAIT_IO,
                           &commandParams,
                           NULL);
    msgCHECK(returnValue);

    /* command an E_STOP_ABORT */
    commandParams.motion.motionCommand  = MPICommandMotionE_STOP_ABORT;
    commandParams.motion.motion     = motion;

    returnValue =
        sequenceCommandAdd(sequence,
                           MPICommandTypeMOTION,
                           &commandParams,
                           NULL);
    msgCHECK(returnValue);

    /* Stop sequence */
    commandParams.branch.label      = NULL;
    commandParams.branch.expr.oper  = MPICommandOperatorALWAYS;

    returnValue =
        sequenceCommandAdd(sequence,
                           MPICommandTypeBRANCH,
                           &commandParams,
                           NULL);
    msgCHECK(returnValue);

    /* Return the new sequence handle */
    return sequence;
}
```

```
/*
   sequenceProgramDelete() removes and deletes all command objects on sequence's
   command list and then deletes sequence.  This function does return MPI error
   codes so that sequence setup code may be more easily debugged.
*/
long sequenceProgramDelete(MPISequence *sequence)
{
    MPICommand  command;
    long        returnValue = MPIMessageOK;


    /* Remove and delete command objects on sequence list */
    while(TRUE) {
        command = mpiSequenceCommandLast(*sequence);
        if ((returnValue != MPIMessageOK) ||
            (command==MPIHandleVOID)) {
            break;
        }
        returnValue =
            mpiSequenceCommandRemove(*sequence,
                                     command);

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiCommandDelete(command);
        }
    }

    /* Delete sequence object */
    returnValue =
        mpiSequenceDelete(*sequence);

    if (returnValue == MPIMessageOK) {
        /* Make it obvious that sequence is no longer a valid object */
        *sequence = MPIHandleVOID;
    }

    return returnValue;
}


/*
   Starts a sequence, waits for the user to press a key and then
   stops the sequence.
*/
void sequenceRun(MPISequence sequence)
{
    long returnValue;


    /* Start sequence */
    returnValue =
        mpiSequenceStart(sequence,
```

```
                        MPIHandleVOID);
    msgCHECK(returnValue);

    fprintf(stderr,"Press any key to stop sequence.\n");
    meiPlatformKey(MPIWaitFOREVER);
    fprintf(stderr,"Exiting ...\n");

    /* Stop sequence */
    returnValue = mpiSequenceStop(sequence);
    if ((returnValue != MPIMessageOK) &&
        (returnValue != MPISequenceMessageSTOPPED)) {
        msgCHECK(returnValue);
    }
}


int main(int     argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;
    MPIAxis             axis[MEIXmpMAX_COORD_AXES];     /* axis handles */
    MPIMotor            motor;
    MPISequence         sequence;

    long    returnValue;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                axis,
                AXIS_COUNT,
                &motor,
                MOTOR_NUMBER);

    /* Create sequence program */
    sequence =
        sequenceKillProgramCreate(motion,
                                  motor,
                                  IO_TRIGGER,
                                  IO_LEVEL);

    /* Run sequence */
```

```
    sequenceRun(sequence);

    /* Delete sequence program */
    returnValue =
        sequenceProgramDelete(&sequence);
    msgCHECK(returnValue);

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   axis,
                   AXIS_COUNT,
                   &motor);

    return ((int)returnValue);
}
```

seqrec.c -- Track status of specified motion supervisors and enable data recorder on any motion.

```
/* seqrec.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/seqrec.c 11    7/23/01 2:36p Kevinh $";
#endif

/*

:Track status of specified motion supervisors and enable data recorder on any motion.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    seqrecMain

argMainRENAME(main, seqrec)
#endif

/* Command line arguments and defaults */
long    sequenceNumber  = 0;

Arg argList[] = {
    {   "-sequence",    ArgTypeLONG,    &sequenceNumber,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPISequence sequence;   /* sequence handle */
```

```
    MPICommand  command;

    MEIXmpData  *firmware;

    MPICommandParams    params;

    long    returnValue;    /* return value from library */

    long    index;

    long            motionCount;
    MPIObjectMap    motionMap;

    long    labelCheckMS;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if (sequenceNumber >= MEIXmpMAX_PSs) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[motion# (0) ...]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_PSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    mpiObjectMapCLEAR(motionMap);

    /* Create motion map */
    if (argIndex >= argc) {
        motionCount = 1;
```

```
        mpiObjectMapBitSET(motionMap, 0, 1);    /* MS[0] default */
    }
    else {
        motionCount = 0;

        while (argIndex < argc) {
            char    *arg;
            long    number;

            arg = argv[argIndex++];

            number = meiPlatformAtol(arg);

            if ((number >= 0) &&
                (number < MEIXmpMAX_MSs)) {
                motionCount++;
                mpiObjectMapBitSET(motionMap, number, 1);
            }
            else {
                fprintf(stderr,
                        "%s: invalid motion number\n",
                        arg);
                exit(1);
            }
        }
    }

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-sequence # (0 .. %d)]\n"
                           "\t\t[motion# (0) ...]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_PSs - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    returnValue =
        mpiControlMemory(control,
                         (void **)&firmware,
                         NULL);
    msgCHECK(returnValue);

    /* Create Sequence */
    sequence =
        mpiSequenceCreate(control,
                          sequenceNumber,
                          2 + motionCount + 2); /* sequence size */
```

```
    msgCHECK(mpiSequenceValidate(sequence));

    /* Create Commands */

    /* Branch to "CheckMS" if Recorder is ON (!= 0) */
    params.branch.label             = "CheckMS";
    params.branch.expr.address.l    = &firmware->Recorder.Enable;
    params.branch.expr.oper         = MPICommandOperatorNOT_EQUAL;
    params.branch.expr.by.value.l   = 0;

    command =
        mpiCommandCreate(MPICommandTypeBRANCH,
                         &params,
                         "RecorderON");
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    /* Turn Recorder ON */
    params.assign.dst.l     = &firmware->Recorder.Enable;
    params.assign.value.l   = -1;   /* continuous, else # records */

    command =
        mpiCommandCreate(MPICommandTypeASSIGN,
                         &params,
                         NULL);
    msgCHECK(mpiCommandValidate(command));

    returnValue =
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);

    labelCheckMS = FALSE;

    for (index = 0; index < MEIXmpMAX_MSs; index++) {
        if (mpiObjectMapBitGET(motionMap, index) == 0) {
            continue;
        }

        /* Branch to "RecorderOn" if motion detected */
        params.branch.label             = "RecorderON";
        params.branch.expr.address.l    = (long *)&firmware->MS[index].Status;
        params.branch.expr.oper         = MPICommandOperatorBIT_CLEAR;
        params.branch.expr.by.value.l   = (0x1 << MEIXmpEventDONE);

        command =
            mpiCommandCreate(MPICommandTypeBRANCH,
                             &params,
                             (labelCheckMS == FALSE) ? "CheckMS" : NULL);
        msgCHECK(mpiCommandValidate(command));

        labelCheckMS = TRUE;

        returnValue =
```

```
        mpiSequenceCommandAppend(sequence,
                                 command);
    msgCHECK(returnValue);
}

/* Turn Recorder OFF */
params.assign.dst.l     = &firmware->Recorder.Enable;
params.assign.value.l   = 0;

command =
    mpiCommandCreate(MPICommandTypeASSIGN,
                     &params,
                     NULL);
msgCHECK(mpiCommandValidate(command));

returnValue =
    mpiSequenceCommandAppend(sequence,
                             command);
msgCHECK(returnValue);

/* Branch to "CheckMS" */
params.branch.label     = "CheckMS";
params.branch.expr.oper = MPICommandOperatorALWAYS;

command =
    mpiCommandCreate(MPICommandTypeBRANCH,
                     &params,
                     NULL);
msgCHECK(mpiCommandValidate(command));

returnValue =
    mpiSequenceCommandAppend(sequence,
                             command);
msgCHECK(returnValue);

/* Start sequence */
returnValue =
    mpiSequenceStart(sequence,
                     MPIHandleVOID);

if (returnValue != MPIMessageOK) {
    fprintf(stderr, "%s: mpiSequenceStart() returns 0x%x: %s\n",
                    argv[0],
                    returnValue,
                    mpiMessage(returnValue, NULL));
    exit(returnValue);
}

meiPlatformConsole("Press any key to stop sequence\n");
meiPlatformKey(MPIWaitFOREVER);

/* Stop sequence */
returnValue = mpiSequenceStop(sequence);
msgCHECK(returnValue);

returnValue = mpiSequenceDelete(sequence);
msgCHECK(returnValue);
```

```
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

settle1.c -- Configure in-position tolerance and settling time for an axis.

```
/* settle1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/settle1.c 10    7/23/01 2:36p Kevinh $";
#endif

/*

:Configure in-position tolerance and settling time for an axis.

A simple settling configuration which sets the Fine Position Tolerance
 and Settling Time for an axis.  Both are required for the
 MEIXmpStatusIN_FINE_POSITION status bit.  The settling criteria are:

    1)  The absolute value of the position error is less than or equal to
        the fine position tolerance.
    2)  The absolute value of the velocity is less than or equal to the
        velocity tolerance.
    3)  Both criteria 1 and 2 above have been satisfied for the settling
        time.  Whenever either criteria 1 or 2 is not satisfied,
        IN_FINE_POSITION is cleared and the settling tier is reset.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    settle1Main

argMainRENAME(main, settle1)
#endif
```

```
/* Command line arguments and defaults */
long    axisNumber  = 0;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

int
    main(int    argc,
         char   *argv[])
{
    MPIControl      control;        /* Motion controller handle */
    MPIAxis         axis;           /* Axis handle(s) */

    MPIAxisConfig   axisConfig;     /* Axis configuration */

    long    returnValue;            /* Return value from library */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber >= MEIXmpMAX_Axes)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
```

```
                            MEIXmpMAX_Axes - 1);
        exit(MPIMessageARG_INVALID);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object on controller*/
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Configure axis */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    axisConfig.inPosition.tolerance.positionFine = (float)50.0;
    axisConfig.inPosition.tolerance.velocity     = (float)100.0;
    axisConfig.inPosition.settlingTime           = (float)0.5; /* .5 seconds */

    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

settle2.c -- Configure settling criteria and exception event settling conditions.

```
/* settle2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/settle2.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Configure settling criteria and exception event settling conditions.

The MPI supports axis configurations for motion settling criteria to
 determine when motion is complete.  The XMP-Series controller supports
 event notification for Coarse, Fine, and Target positions and motion done.
 Generally, settling criteria is only applicable to normal host commanded
 motion profiles.  The XMP-Series controller can also be configured to
 support settling criteria with Stop and E-Stop events.

The settling criteria is configured set with mpiAxisConfigGet/Set(...),
 using the structures:

typedef struct MPIAxisInPosition {
    struct {
        float   positionFine;
        long    positionCoarse;
        float   velocity;
    } tolerance;
    float   settlingTime;
    long    settleOnStop;
    long    settleOnEstop;
} MPIAxisInPosition;

typedef struct MPIAxisConfig {
    MPIAxisInPosition   inPosition;
    MPIObjectMap    filterMap;
} MPIAxisConfig;


The settling criteria are:

 positionFine - in-position window between command and actual (counts)
```

```
 positionCoarse - in-position window between target and actual (counts)

 velocity - actual velocity (counts/second)

 settlingTime - required time for settling (seconds)


The Stop and E-Stop event settling conditions can be enabled/disabled:

 settleOnStop - TRUE or FALSE, enables/disables settling criteria for Stop

 settleOnEstop - TRUE or FALSE, enables/disables settling criteria for E-Stop


The XMP-Series controller sets the appropriate status bits and generates
 the following events based on the settling criteria and logic:

 IN_FINE_POSITION

    1) |command position - actual position| < positionFine tolerance
    2) |command velocity - actual velocity| < velocity tolerance
    3) 1,2 and 3 have been satisfied for the settlingTime tolerance

 IN_COARSE_POSITION

    1) |target position - actual position| < postionCoarse tolerance

 AT_TARGET

    1) The command position = target position.

 DONE

    1) The calculated trajectory is complete.
    2) IN_FINE_POSITION conditions have been satisfied.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    settle2Main

argMainRENAME(main, settle2)
#endif
```

```
#define MOTION_COUNT        (1)     /* Number of motion profiles */
#define AXIS_COUNT          (1)     /* Number of axes */


/* Command line arguments and defaults */
long            axisNumber[AXIS_COUNT] = { 0,   };
long            motionNumber    = 0;
MPIMotionType   motionType      = MPIMotionTypeS_CURVE;
float           fineTol         = (float)500.0; /* counts */
long            coarseTol       = 200;          /* counts */
float           velocityTol     = (float)1.0;   /* counts/sec */
float           settlingTime    = (float)0.2;   /* seconds */
long            eventDelay      = 1000;         /* milliseconds */

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber[0], },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-type",    ArgTypeLONG,    &motionType,    },
    {   "-fine",    ArgTypeFLOAT,   &fineTol,       },
    {   "-coarse",  ArgTypeLONG,    &coarseTol,     },
    {   "-velocity",ArgTypeFLOAT,   &velocityTol,   },
    {   "-settling",ArgTypeFLOAT,   &settlingTime,  },
    {   "-delay",   ArgTypeLONG,    &eventDelay,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[][AXIS_COUNT] = {
    { 20000.0,  },
    { 0.0,  },
};

MPITrajectory trajectory[][AXIS_COUNT] = {
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
    {   /* velocity     accel       decel       jerkPercent */
        { 10000.0,      1000000.0,  1000000.0,  0.0,    },
    },
};

/* Motion parameters */

MPIMotionSCurve sCurve[] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};

MPIMotionTrapezoidal    trapezoidal[] = {
    {   &trajectory[0][0],  &position[0][0],    },
    {   &trajectory[1][0],  &position[1][0],    },
};
```

```
MPIMotionVelocity   velocity[] = {
    {   &trajectory[0][0],  },
    {   &trajectory[1][0],  },
};

#define SETTLE_EVENTS   (2) /* Number of settle configurations */

/* Settle on Stop/E-Stop Events */
long settleEvent[] = {
    FALSE,
    TRUE
};

#define ACTIONS         (5) /* Number of MPI actions */

/* Generate Actions */
MPIAction action[] = {
    MPIActionNONE,
    MPIActionSTOP,
    MPIActionE_STOP,
    MPIActionE_STOP_ABORT,
    MPIActionABORT
};

char * actionString[] = {
    "None",
    "Stop",
    "E-Stop",
    "E-Stop/Abort",
    "Abort"
};

void eventStatusDisplay(MPIEventStatus status);


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* Motion controller handle */
    MPIAxis     axis;       /* Axis object */
    MPIMotion   motion;     /* Motion object */
    MPINotify   notify;     /* Event notification object */
    MPIEventMgr eventMgr;   /* Event manager handle */

    MPIAxisConfig   axisConfig; /* Axis configuration object */

    MPIEventMask    eventMask;

    long    returnValue;    /* Return value from library */

    long    index;
    long    actionIndex;    /* Index to action[...] array */
    long    settleIndex;    /* Index to settleEvent[...] array */
```

```
    long    motionDone;     /* Flag when Done occurs */
    long    previousEvents; /* Flag to check for previously generated events */

    double  actual;         /* Axis position */
    double  origin;         /* Axis origin */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber[0] > (MEIXmpMAX_Axes - AXIS_COUNT)) ||
        (motionNumber  >= MEIXmpMAX_MSs) ||
        (motionType <  MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST) ||
        (fineTol < 0.0) ||
        (coarseTol < 0) ||
        (velocityTol < 0.0) ||
        (settlingTime < 0.0) ||
        (eventDelay < 0)) {
        printf("usage: %s %s\n"
               "\t\t[-axis # (0 .. %d)]\n"
               "\t\t[-motion # (0 .. %d)]\n"
               "\t\t[-type # (0 .. %d)]\n"
               "\t\t[-fine # (0.0 .. )]\n"
               "\t\t[-coarse # (0 .. )]\n"
               "\t\t[-velocity # (0.0 .. )]\n"
               "\t\t[-settling # (0.0 .. )]\n"
```

```
            "\t\t[-delay # (0 .. )]\n",
             argv[0],
             ArgUSAGE,
             MEIXmpMAX_Axes - AXIS_COUNT,
             MEIXmpMAX_MSs - 1,
             MEIMotionTypeLAST - 1);
    exit(MPIMessageARG_INVALID);
}

switch (motionType) {
    case MPIMotionTypeS_CURVE:
    case MPIMotionTypeTRAPEZOIDAL:
    case MPIMotionTypeVELOCITY: {
        break;
    }
    default: {
        meiPlatformConsole("%s: %d: motion type not available\n",
                           argv[0],
                           motionType);
        exit(MPIMessageUNSUPPORTED);
        break;
    }
}

/* Create motion controller object */
control =
    mpiControlCreate(controlType,
                     &controlAddress);
msgCHECK(mpiControlValidate(control));

/* Initialize motion controller */
returnValue = mpiControlInit(control);
msgCHECK(returnValue);

/* Create axis object for axisNumber */
axis =
    mpiAxisCreate(control,
                  axisNumber[0]);
msgCHECK(mpiAxisValidate(axis));

/* Create motion object, appending the axis object */
motion =
    mpiMotionCreate(control,
                    motionNumber,
                    axis);
msgCHECK(mpiMotionValidate(motion));

/* Request notification of ALL events from motion */
mpiEventMaskCLEAR(eventMask);
mpiEventMaskALL(eventMask);
meiEventMaskALL(eventMask);

returnValue =
    mpiMotionEventNotifySet(motion,
```

```
                                eventMask,
                                NULL);      /* ALL sources */
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    meiPlatformConsole("Press any key to quit ...\n");

    /* Loop repeatedly */
    index       = 0;
    actionIndex = 0;
    settleIndex = 0;
    motionDone  = TRUE;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus      eventStatus;
        MPIStatus           status;

        if (motionDone) {
            MPIMotionParams     motionParams;       /* Motion parameters */

            /* Fill in the MPIMotionParams structure */
            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
```

```
        }
        default: {
            meiASSERT(FALSE);
            break;
        }
    }

    /* Check motion status for any error conditions */
    returnValue =
        mpiMotionStatus(motion,
                        &status,
                        NULL);
    msgCHECK(returnValue);

    if (status.state == MPIStateERROR) {
        returnValue =
            mpiMotionAction(motion,
                            MPIActionRESET);
        msgCHECK(returnValue);

        printf("  Error state cleared.\n");
    }

    /* Set origin to current actual position */
    returnValue =
        mpiAxisActualPositionGet(axis,
                                 &actual);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisOriginGet(axis,
                         &origin);
    msgCHECK(returnValue);

    returnValue =
        mpiAxisOriginSet(axis,
                         (origin + actual));
    msgCHECK(returnValue);

    /* Configure axis settling criteria */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    axisConfig.inPosition.tolerance.positionFine =
        fineTol;
    axisConfig.inPosition.tolerance.positionCoarse =
        coarseTol;
    axisConfig.inPosition.tolerance.velocity =
        velocityTol;
    axisConfig.inPosition.settlingTime =
        settlingTime;
```

```
/* Configure settling criteria for Stop and E-Stop events
    TRUE = use settling criteria for Stop and E-Stop
    FALSE = do NOT use settling criteria for Stop and E-Stop
*/
axisConfig.inPosition.settleOnStop = settleEvent[settleIndex];
axisConfig.inPosition.settleOnEstop = settleEvent[settleIndex];

returnValue =
    mpiAxisConfigSet(axis,
                     &axisConfig,
                     NULL);
msgCHECK(returnValue);

/* Check for any previous events (generated by RESET) */
previousEvents = TRUE;
while (previousEvents) {
    returnValue =
        mpiNotifyEventWait(notify,
                           &eventStatus,
                           MPIWaitPOLL);
    if (returnValue == MPIMessageTIMEOUT) {
        previousEvents = FALSE;
    }
    else {
        msgCHECK(returnValue);
        eventStatusDisplay(eventStatus);
    }
}

meiPlatformConsole("\nMotion Start, Target= %12.0lf  ",
       position[index][axisNumber[0]]);

meiPlatformConsole("  Settle on Event: %s",
       settleEvent[settleIndex] ? "yes" : "no");

/* Point to point motion (relative to new origin) */
returnValue =
    mpiMotionStart(motion,
                   motionType,
                   &motionParams);
msgCHECK(returnValue);

/* Wait for motion profile to partially execute */
meiPlatformSleep(eventDelay);

/* Generate an event */
returnValue =
    mpiMotionAction(motion,
                    action[actionIndex]);
msgCHECK(returnValue);

meiPlatformConsole("  Event:%s.\n", actionString[actionIndex]);
```

```
            /* Increment through settleEvent[...] and action[...] */
            if (++actionIndex >= ACTIONS) {
                actionIndex = 0;
                if (++settleIndex >= SETTLE_EVENTS) {
                    settleIndex = 0;
                }
            }
            motionDone = FALSE;
        }

        /* Wait for motion event */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               MPIWaitFOREVER);
        msgCHECK(returnValue);

        /* Motion Done Event from motion source */
        if  (eventStatus.type == MPIEventTypeMOTION_DONE) {
            motionDone = TRUE;

            if (++index >= MOTION_COUNT) {
                index = 0;
            }
        }
        eventStatusDisplay(eventStatus);
    }

    printf("\n");

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}


void eventStatusDisplay(MPIEventStatus status)
{
```

```
    MEIEventStatusInfo  *info;

    info = (MEIEventStatusInfo *)status.info;

    switch (status.type) {
        /* In Coarse Event from axis source */
        case MEIEventTypeIN_POSITION_COARSE: {
            meiPlatformConsole("  InCoarse, actual:%ld, sample:0x%x\n",
                   info->data.axis.actualPosition,
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeIN_POSITION_FINE: {
            meiPlatformConsole("  InFine, actual:%ld, sample:0x%x\n",
                   info->data.axis.actualPosition,
                   info->data.axis.sampleCounter);
            break;
        }
        /* At Target Event from axis source */
        case MEIEventTypeAT_TARGET: {
            meiPlatformConsole("  AtTarget, actual:%ld, sample:0x%x\n",
                   info->data.axis.actualPosition,
                   info->data.axis.sampleCounter);
            break;
        }
        /* Motion Done Event from motion source */
        case MPIEventTypeMOTION_DONE: {
            meiPlatformConsole("  Done, sample:0x%x\n",
                   info->data.motion.sampleCounter);
            break;
        }
        default: {
            meiPlatformConsole("Event (unexpected): %ld (0x%x)",
                   status.type,
                   info->data.word[0]);
            break;
        }
    }
}
```

shape.c -- Configure Trajectory Shaping Filters

```
/* shape.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/shape.c 12    7/23/01 2:36p Kevinh $";
#endif

/*

:Configure Trajectory Shaping Filters

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    shapeMain

argMainRENAME(main, shape)
#endif

typedef struct SHAPER {
    long    Index;
    long    Axis;
    long    Length;
    long    MaxDelay;
    long    MaxAmp;
    double  TimeStep;
    long    Time[MEIXmpMAX_PreCoeffs/2];
    long    Amp[MEIXmpMAX_PreCoeffs/2];
} SHAPER;
```

```
long
    loadShaper(MPIControl   control,
               SHAPER       *shaper);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl          control;    /* motion controller handle */
    MPIControlConfig    config;

    SHAPER      shaper;

    char    *fileName;
    FILE    *fp;
    char    buffer[256];
    long    shaperCount;
    long    shaperCountOld;

    long    returnValue;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    fileName =
        (argIndex >= argc)
                ? "shape.txt"
                : argv[argIndex++];

    if (argIndex < argc) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[fileName (%s)]\n",
                           argv[0],
                           ArgUSAGE,
                           "shape.txt");
        exit(0);
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);
```

```
    fp = fopen(fileName, "r");

    if (fp == NULL) {
        fprintf(stderr,
                "shape.txt not found.\n");
        exit(2);
    }

    returnValue =
        mpiControlConfigGet(control,
                            &config,
                            NULL);
    msgCHECK(returnValue);

    shaperCount = 0;
    shaperCountOld = -1;

    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        long    axis;
        long    length;
        long    maxAmp;
        long    maxDelay;

        char    *ptr;

        if (shaperCount > shaperCountOld) {
            shaper.Axis     = -1;   /* Make sure all params are read from file */
            shaper.MaxAmp   = -1;
            shaper.Length   =  0;
            shaper.MaxDelay = -1;
            shaper.TimeStep = 1.0 / config.sampleRate;

            shaperCountOld = shaperCount;
        }

        for (ptr = buffer; *ptr != '\0'; ptr++) {
            char    byte = *ptr;

            if (isupper(byte)) {
                *ptr = (char)tolower(byte);
            }
        }

        if (sscanf(buffer, "#AXIS  = %d", &axis) == 1) {
            shaper.Axis = axis;
            continue;
        }

        if (sscanf(buffer, "#IMPULSES = %d", &length) == 1) {
            shaper.Length = length;
            continue;
        }

        if (sscanf(buffer, "#AMPSUM = %d", &maxAmp) == 1) {
            shaper.MaxAmp = maxAmp;
            continue;
```

```
        }

        if (sscanf(buffer, "#STEPS = %d", &maxDelay) == 1) {
            shaper.MaxDelay = maxDelay;
            continue;
        }

        if (strncmp(buffer, "#TIME", 5) == 0) {
            long    sum;
            long    maxTime;
            long    index;

            sum = 0;
            maxTime = 0;

            for (index = 0; index < shaper.Length; index++) {
                long    time;
                long    amp;

                if (fgets(buffer, sizeof(buffer), fp) == NULL) {
                    fprintf(stderr,
                            "Unexpected end-of-file\n");
                    return ((int)returnValue);

                }

                if (sscanf(buffer, "%d %d", &time, &amp) == 2) {
                    shaper.Time[index]  = time;
                    shaper.Amp[index]   = amp;
                    sum += amp;

                    if (time > maxTime) {
                        maxTime = time;
                    }
                }
            }

            if (sum != shaper.MaxAmp) {
                fprintf(stderr,
                        "Sum of amplitudes (%d) does not equal AmpSum (%d).\n",
                        sum,
                        shaper.MaxAmp);
                return ((int)returnValue);
            }

            if (maxTime != shaper.MaxDelay) {
                fprintf(stderr,
                        "Maximum time (%d) does not equal Steps (%d).\n",
                        maxTime,
                        shaper.MaxDelay);
                return (returnValue);
            }

            if (axis < 0) {
                fprintf(stderr,
                        "No #Axis statement.\n");
```

```
                return (returnValue);
            }
            else {
                shaper.Index = shaperCount;

                if (shaperCount >= MEIXmpMAX_PreFilters) {
                    fprintf(stderr,
                            "Too many shapers.\n");
                    break;
                }
                else {
                    double  scale;

                    /* Rescale for XMP filter normalization */
                    scale = (double)MEIXmpFILTER_SCALE / (double)shaper.MaxAmp;

                    sum = 0;

                    for (index = 0; index < shaper.Length - 1; index++) {
                        shaper.Amp[index] = (long)(shaper.Amp[index] * scale);
                        sum += shaper.Amp[index];
                    }
                    shaper.Amp[shaper.Length - 1] = MEIXmpFILTER_SCALE - sum;

                    loadShaper(control,
                               &shaper);

                    fprintf(stderr,
                            "Axis %d shaper loaded.\n",
                            axis);

                    shaperCount++;
                }
            }
        }
    }
    fclose(fp);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long
    loadShaper(MPIControl   control,
               SHAPER       *shaper)
{
    MEIXmpData          *firmware;
    MEIXmpBufferData    *external;

    MPIAxis axis;

    MPIControlConfig    controlConfig;
    MEIControlConfig    controlConfigXmp;
    MPIAxisConfig       axisConfig;
```

```
    MEIAxisConfig       axisConfigXmp;

    long    returnValue;

    long    index;
    long    axisNumber;
    long    length;
    long    delayTime;

    /* Get pointer to XMP firmware */
    returnValue =
        mpiControlMemory(control,
                         (void **)&firmware,
                         (void **)&external);
    msgCHECK(returnValue);

    index      = shaper->Index;
    axisNumber = shaper->Axis;

    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);

    /* Set command position to Filter input (disable filter) */
    axisConfigXmp.Filter.Pointer    = &firmware->Axis[axisNumber].TC.Filter.Input;
    axisConfigXmp.Filter.Delay      = 0;

    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);

    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            &controlConfigXmp);
    msgCHECK(returnValue);

    controlConfigXmp.PreFilter[index].Type      = MEIXmpFilterTypeSHAPING;
    controlConfigXmp.PreFilter[index].Length    = shaper->Length;
    controlConfigXmp.PreFilter[index].Input     =
&firmware->Axis[axisNumber].TC.Filter.Input;
    controlConfigXmp.PreFilter[index].Coeff[0]  = 0;

    for (length = 0; length < shaper->Length; length++) {
        controlConfigXmp.PreFilter[index].Coeff[2 * length + 1] =
            shaper->Time[length];
        controlConfigXmp.PreFilter[index].Coeff[2 * length + 2] =
```

```
            shaper->Amp[length];
    }

    if (controlConfigXmp.preFilterCount < (index+1)) {
        controlConfigXmp.preFilterCount = (index + 1);
    }

    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            &controlConfigXmp);
    msgCHECK(returnValue);

    /* 2x max delay */
    delayTime = 2 * (long)(shaper->MaxDelay * shaper->TimeStep * 1000.0);

    meiPlatformSleep(delayTime);    /* wait for filter to initialize */

    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);

    /* Set command position pointer to Output of shaper */
    axisConfigXmp.Filter.Pointer    = &external->PreFilter[index].Output;
    axisConfigXmp.Filter.Delay      = shaper->MaxDelay;

    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         &axisConfigXmp);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    return (returnValue);
}
```

sidn1.c -- SERCOS node idn get/display

```
/* sidn1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/sidn1.c 16    7/23/01 2:36p Kevinh $";
#endif

/*

:SERCOS node idn get/display

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    sidn1Main

argMainRENAME(main, sidn1)
#endif

#define BAUD_RATE_DEFAULT       (MPISercosBaud10MBIT)
#define PHASE_DEFAULT           (2)
#define SAMPLE_RATE_DEFAULT     (1000)
#define MODE_DEFAULT            (MPINodeModeOPENLOOP_POSITION_MOTOR)
#define INTENSITY_DEFAULT       (3)
#define SERCOS_NUMBER_DEFAULT   (0)
#define NODE_NUMBER_DEFAULT     (0)

/* Command line arguments and defaults */
long            baud            = -1;
long            phase           = PHASE_DEFAULT;
```

```
long            sampleRate      = SAMPLE_RATE_DEFAULT;
MPINodeMode     mode            = MODE_DEFAULT;
long            intensity       = INTENSITY_DEFAULT;
long            sercosNumber    = SERCOS_NUMBER_DEFAULT;
long            nodeNumber      = NODE_NUMBER_DEFAULT;

Arg argList[] = {
    {   "-baud",        ArgTypeLONG,    &baud,          },
    {   "-phase",       ArgTypeLONG,    &phase,         },
    {   "-sample",      ArgTypeLONG,    &sampleRate,    },
    {   "-mode",        ArgTypeLONG,    &mode,          },
    {   "-intensity",   ArgTypeLONG,    &intensity,     },
    {   "-sercos",      ArgTypeLONG,    &sercosNumber,  },
    {   "-node",        ArgTypeLONG,    &nodeNumber,    },

    {   NULL,           ArgTypeINVALID, NULL,   }
};

#if 0
#define MEI_PMC
#endif

void
    usage(char  *programName)
{
    meiPlatformConsole("usage: %s\n\t%s\n"
                       "\t\t[-baud (%d)]\n"
                       "\t\t[-phase (%d)]\n"
                       "\t\t[-sample (%d)]\n"
                       "\t\t[-mode (%d)]\n"
                       "\t\t[-intensity (%d)]\n"
                       "\t\t[-sercos (%d)]\n"
                       "\t\t[-node (%d)]\n"
                       "\t\tidn [...]\n",
                       programName,
                       ArgUSAGE,
                       (BAUD_RATE_DEFAULT == MPISercosBaud2MBIT)  ?  2 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud4MBIT)  ?  4 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud10MBIT) ? 10
                                                                  : -1,
                       PHASE_DEFAULT,
                       SAMPLE_RATE_DEFAULT,
                       MODE_DEFAULT,
                       INTENSITY_DEFAULT,
                       SERCOS_NUMBER_DEFAULT,
                       NODE_NUMBER_DEFAULT);

    exit(0);
}

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;
```

```
    MPISercos   sercos = MPIHandleVOID;
    MPINode     node   = MPIHandleVOID;
    MPIIdn      idn    = MPIHandleVOID;

    long    returnValue;
    long    deleteMessage;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPISercosStatus     SercosStatus;

    long    argIndex;

    MPISercosBaud   baudRate;
    MPISercosConfig sercosConfig;

    MPISercosStatus sercosStatus;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if (argIndex >= argc) {
        usage(argv[0]);
    }

    baudRate =
        (baud == 2)  ? MPISercosBaud2MBIT  :
        (baud == 4)  ? MPISercosBaud4MBIT  :
        (baud == 10) ? MPISercosBaud10MBIT
                     : BAUD_RATE_DEFAULT;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
```

```
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create sercos object */
    sercos =
        mpiSercosCreate(control,
                        sercosNumber);
    msgCHECK(mpiSercosValidate(sercos));

    returnValue =
        mpiSercosStatus(sercos,
                        &sercosStatus,
                        NULL);
    msgCHECK(returnValue);

    printf("sercos[%d] status: phase %d\n",
            sercosNumber,
            sercosStatus.phase);

    if (sercosStatus.phase < 2) {
        MPIControlConfig    controlConfig;

        printf("\nsercos[%d]: transitioning to phase 2\n",
                sercosNumber);

        returnValue =
            mpiControlConfigGet(control,
                                &controlConfig,
                                NULL);
        msgCHECK(returnValue);

        controlConfig.sampleRate = sampleRate;

#if defined(MEI_PMC)
        controlConfig.axisCount     = 8;
        controlConfig.filterCount   = 8;
        controlConfig.motionCount   = 8;
        controlConfig.motorCount    = 8;
        controlConfig.sequenceCount = 8;

        controlConfig.sercosCount = 1;
#else
        controlConfig.axisCount     = 24;
        controlConfig.filterCount   = 24;
        controlConfig.motionCount   = 24;
        controlConfig.motorCount    = 24;
        controlConfig.sequenceCount = 24;

        controlConfig.sercosCount = 3;
#endif

        returnValue =
```

```
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    returnValue =
        mpiSercosConfigGet(sercos,
                           &sercosConfig,
                           NULL);
    msgCHECK(returnValue);

    sercosConfig.baudRate = baudRate;
    sercosConfig.xmitIntensity = intensity;

    returnValue =
        mpiSercosConfigSet(sercos,
                           &sercosConfig,
                           NULL);
    msgCHECK(returnValue);

    returnValue =
        mpiSercosInit(sercos,
                      2);
    msgCHECK(returnValue);

    returnValue =
        mpiSercosStatus(sercos,
                        &SercosStatus,
                        NULL);
    msgCHECK(returnValue);

    printf("sercos[%d] status: phase %d\n",
            sercosNumber,
            SercosStatus.phase);
}

node =
    mpiNodeCreate(sercos,
                  nodeNumber);
msgCHECK(mpiNodeValidate(node));

returnValue =
    mpiSercosNodeAppend(sercos,
                        node);
msgCHECK(returnValue);



idn = mpiIdnCreate((MPIIdnNumber)(mpiIdnNumberSTANDARD(1)));
returnValue = mpiIdnValidate(idn);

while ((returnValue == MPIMessageOK) &&
       (argIndex < argc)) {
    MPIIdnElement   idnElement;
    MPIIdnField     field;
```

```
        long    idnNumber;

        char    idnText[32];

        idnNumber = meiPlatformAtol(argv[argIndex++]);

        returnValue =
            mpiIdnNumberSET(idn,
                            &idnNumber);

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiNodeIdnGET(node,
                              idn);
        }

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiIdnElementGET(idn,
                                 &idnElement);
        }

        if (returnValue == MPIMessageOK) {
            printf("\nsercos[%d] node number %d idn %s:\n",
                    sercosNumber,
                    nodeNumber,
                    mpiIdnNumberName((MPIIdnNumber)idnNumber, idnText));

            for (field = MPIIdnFieldFIRST; field < MPIIdnFieldLAST; field++) {
                switch (field) {
                    case MPIIdnFieldNUMBER: {
                        const char  *text;

                        if (returnValue == MPIMessageOK) {
                            returnValue =
                                mpiIdnNumberText(idn,
                                                 &text);
                        }

                        if (returnValue == MPIMessageOK) {
                            printf("\tnumber %s\n",
                                    text);
                        }
                        break;
                    }
                    case MPIIdnFieldNAME: {
                        printf("\tname %s\n",
                                idnElement.name);
                        break;
                    }
                    case MPIIdnFieldATTRIBUTES: {
                        printf("\tattributes 0x%x\n",
                                idnElement.attributes);
                        break;
```

```
                    }
                    case MPIIdnFieldUNIT: {
                        printf("\tunit %s\n",
                                idnElement.unit);
                        break;
                    }
                    case MPIIdnFieldMINIMUM: {
                        printf("\tminimum 0x%x\n",
                                idnElement.minimum.u);
                        break;
                    }
                    case MPIIdnFieldMAXIMUM: {
                        printf("\tmaximum 0x%x\n",
                                idnElement.maximum.u);
                        break;
                    }
                    case MPIIdnFieldDATA: {
                        if (mpiIdnFieldAttrGET(MPIIdnFieldAttrVARIABLE,
idnElement.attributes) == 0) {
                            printf("\tdata 0x%x\n",
                                    idnElement.data.binary);
                        }
                        else {
                            MPIIdnVarLength *varLength;

                            long    index;

                            varLength = &idnElement.data.varLength;

                            for (index = 0; index < varLength->count; index++) {
                                printf("\tdata[%d] ",
                                        index);

                                switch (varLength->type) {
                                    case MPIIdnDataTypeBINARY: {
                                        printf("0x%x\n",
                                                varLength->as.value[index].u);
                                        break;
                                    }
                                    case MPIIdnDataTypeIDN: {
                                        char    idnName[32];

                                        printf("%s\n",

mpiIdnNumberName(varLength->as.idn[index],
                                                                        idnName));
                                        break;
                                    }
                                    case MPIIdnDataTypeLONG: {
                                        printf("%d\n",
                                                varLength->as.value[index].l);
                                        break;
                                    }
                                    case MPIIdnDataTypeTEXT: {
                                        printf("%s\n",
```

```
                                        varLength->as.text);
                                    break;
                                }
                                case MPIIdnDataTypeUNSIGNED: {
                                    printf("%u\n",
                                        varLength->as.value[index].u);
                                    break;
                                }
                                case MPIIdnDataTypeUNSIGNED_HEX: {
                                    printf("0x%x\n",
                                        varLength->as.value[index].u);
                                    break;
                                }
                                default: {
                                    break;
                                }
                            }

                            if (varLength->type == MPIIdnDataTypeTEXT) {
                                break;
                            }
                        }
                    }
                    break;
                }
                default: {
                    break;
                }
            }
        }
    }
}

/* Delete the IDN handle */
if (idn != MPIHandleVOID) {
    deleteMessage = mpiIdnDelete(idn);

    if (returnValue == MPIMessageOK) {
        returnValue = deleteMessage;
    }
}

/* Clear Sercos Node list */
if (sercos != MPIHandleVOID) {
    deleteMessage =
        mpiSercosNodeListSet(sercos,
                             0,
                             NULL);

    if (returnValue == MPIMessageOK) {
        returnValue = deleteMessage;
    }
}

/* Delete the NODE handle */
```

```
    if (node != MPIHandleVOID) {
        deleteMessage = mpiNodeDelete(node);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Delete the SERCOS handle */
    if (sercos != MPIHandleVOID) {
        deleteMessage = mpiSercosDelete(sercos);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Delete the CONTROL handle */
    if (control != MPIHandleVOID) {
        deleteMessage = mpiControlDelete(control);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    return ((int)returnValue);
}
```

sidn2.c -- SERCOS node idn get/display/set

```
/* sidn2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/sidn2.c 18    7/23/01 2:36p Kevinh $";
#endif

/*

:SERCOS node idn get/display/set

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    sidn2Main

argMainRENAME(main, sidn2)
#endif

#define BAUD_RATE_DEFAULT       (MPISercosBaud10MBIT)
#define PHASE_DEFAULT           (2)
#define SAMPLE_RATE_DEFAULT     (1000)
#define MODE_DEFAULT            (MPINodeModeOPENLOOP_POSITION_MOTOR)
#define INTENSITY_DEFAULT       (3)
#define SERCOS_NUMBER_DEFAULT   (0)
#define NODE_NUMBER_DEFAULT     (0)
```

```
/* Command line arguments and defaults */
long            baud            = -1;
long            phase           = PHASE_DEFAULT;
long            sampleRate      = SAMPLE_RATE_DEFAULT;
MPINodeMode     mode            = MODE_DEFAULT;
long            intensity       = INTENSITY_DEFAULT;
long            sercosNumber    = SERCOS_NUMBER_DEFAULT;
long            nodeNumber      = NODE_NUMBER_DEFAULT;

Arg argList[] = {
    {   "-baud",        ArgTypeLONG,    &baud,          },
    {   "-phase",       ArgTypeLONG,    &phase,         },
    {   "-sample",      ArgTypeLONG,    &sampleRate,    },
    {   "-mode",        ArgTypeLONG,    &mode,          },
    {   "-intensity",   ArgTypeLONG,    &intensity,     },
    {   "-sercos",      ArgTypeLONG,    &sercosNumber,  },
    {   "-node",        ArgTypeLONG,    &nodeNumber,    },

    {   NULL,           ArgTypeINVALID, NULL,   }
};

#if 0
#define MEI_PMC
#endif

unsigned long VarData[64];

long
    idnDisplay(MPINode          node,
               MPIIdn           idn,
               unsigned long    *attributes);

void
    usage(char  *programName)
{
    meiPlatformConsole("usage: %s\n\t%s\n"
                       "\t\t[-baud (%d)]\n"
                       "\t\t[-phase (%d)]\n"
                       "\t\t[-sample (%d)]\n"
                       "\t\t[-mode (%d)]\n"
                       "\t\t[-intensity (%d)]\n"
                       "\t\t[-sercos (%d)]\n"
                       "\t\t[-node (%d)]\n"
                       "\t\tidn [...]\n",
                       programName,
                       ArgUSAGE,
                       (BAUD_RATE_DEFAULT == MPISercosBaud2MBIT)  ?  2 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud4MBIT)  ?  4 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud10MBIT) ? 10
                                                                  : -1,
                       PHASE_DEFAULT,
                       SAMPLE_RATE_DEFAULT,
                       MODE_DEFAULT,
```

```
                        INTENSITY_DEFAULT,
                        SERCOS_NUMBER_DEFAULT,
                        NODE_NUMBER_DEFAULT);

    exit(MPIMessageARG_INVALID);
}

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;
    MPISercos   sercos = MPIHandleVOID;
    MPINode     node   = MPIHandleVOID;
    MPIIdn      idn    = MPIHandleVOID;

    long    returnValue;

    long    deleteMessage;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPISercosStatus     SercosStatus;

    long    argIndex;

    MPIIdnNumber    idnNumber;
    MPIIdnData      idnData;

    MPISercosBaud   baudRate;
    MPISercosConfig sercosConfig;

    MPISercosStatus sercosStatus;

    unsigned long   attributes;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
```

```
        else {
            argIndex = argIndexNew;
        }
    }

    if (argIndex >= argc) {
        usage(argv[0]);
    }

    idnNumber = (MPIIdnNumber)(meiPlatformAtol(argv[argIndex++]));

    /* Check for unknown/invalid command line arguments */
    if (argIndex >= argc) {
        usage(argv[0]);
    }

    baudRate =
        (baud == 2)  ? MPISercosBaud2MBIT  :
        (baud == 4)  ? MPISercosBaud4MBIT  :
        (baud == 10) ? MPISercosBaud10MBIT
                     : BAUD_RATE_DEFAULT;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create sercos object */
    sercos =
        mpiSercosCreate(control,
                        sercosNumber);
    msgCHECK(mpiSercosValidate(sercos));

    returnValue =
        mpiSercosStatus(sercos,
                        &sercosStatus,
                        NULL);
    msgCHECK(returnValue);

    printf("sercos[%d] status: phase %d\n",
            sercosNumber,
            sercosStatus.phase);

    if (sercosStatus.phase < 2) {
        MPIControlConfig    controlConfig;

        printf("\nsercos[%d]: transitioning to phase 2\n",
                sercosNumber);
```

```
        returnValue =
            mpiControlConfigGet(control,
                                &controlConfig,
                                NULL);
        msgCHECK(returnValue);

        controlConfig.sampleRate = sampleRate;

#if defined(MEI_PMC)
        controlConfig.axisCount     = 8;
        controlConfig.filterCount   = 8;
        controlConfig.motionCount   = 8;
        controlConfig.motorCount    = 8;
        controlConfig.sequenceCount = 8;

        controlConfig.sercosCount = 1;
#else
        controlConfig.axisCount     = 24;
        controlConfig.filterCount   = 24;
        controlConfig.motionCount   = 24;
        controlConfig.motorCount    = 24;
        controlConfig.sequenceCount = 24;

        controlConfig.sercosCount = 3;
#endif

        returnValue =
            mpiControlConfigSet(control,
                                &controlConfig,
                                NULL);
        msgCHECK(returnValue);

        returnValue =
            mpiSercosConfigGet(sercos,
                               &sercosConfig,
                               NULL);
        msgCHECK(returnValue);

        sercosConfig.baudRate = baudRate;
        sercosConfig.xmitIntensity = intensity;

        returnValue =
            mpiSercosConfigSet(sercos,
                               &sercosConfig,
                               NULL);
        msgCHECK(returnValue);

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiSercosInit(sercos,
                              2);
        }
```

```
    returnValue =
        mpiSercosStatus(sercos,
                        &SercosStatus,
                        NULL);
    msgCHECK(returnValue);

    printf("sercos[%d] status: phase %d\n",
            sercosNumber,
            SercosStatus.phase);
}

node =
    mpiNodeCreate(sercos,
                  nodeNumber);
msgCHECK(mpiNodeValidate(node));

returnValue =
    mpiSercosNodeAppend(sercos,
                        node);
msgCHECK(returnValue);

idn = mpiIdnCreate(idnNumber);
msgCHECK(mpiIdnValidate(idn));

printf("\nsercos[%d] nodeNumber %d idnNumber %d:\n",
        sercosNumber,
        nodeNumber,
        idnNumber);

returnValue =
    idnDisplay(node,
               idn,
               &attributes);
msgCHECK(returnValue);

if (mpiIdnFieldAttrGET(MPIIdnFieldAttrVARIABLE, attributes) == 0) {
    idnData.binary = strtoul(argv[argIndex++], NULL, 0);

    if (argIndex < argc) {
        usage(argv[0]);
    }
}
else {
    MPIIdnVarLength *varLength;

    varLength = &idnData.varLength;

    varLength->type = mpiIdnFieldAttrDATA_TYPE(attributes);

    switch (varLength->type) {
        case MPIIdnDataTypeTEXT: {
            varLength->as.text = argv[argIndex++];
```

```
                varLength->count   = strlen(varLength->as.text);

                if (argIndex < argc) {
                    usage(argv[0]);
                }
                break;
            }
            case MPIIdnDataTypeINVALID: {
                meiASSERT(FALSE);
                break;
            }
            default: {
                long    index;

                varLength->as.value = (MPIIdnValue *)VarData;
                varLength->count    = argc - argIndex;

                for (index = 0; index < varLength->count; index++) {
                    VarData[index] = strtoul(argv[argIndex++], NULL, 0);
                }
                break;
            }
        }
        }

    returnValue =
        mpiIdnDataSET(idn,
                      &idnData);
    msgCHECK(returnValue);

    returnValue =
        mpiNodeIdnDataSET(node,
                          idn);

    if (returnValue == MPIMessageOK) {
        returnValue =
            idnDisplay(node,
                       idn,
                       &attributes);
    }
    else {
        printf("Error 0x%x: %s\n",
                returnValue,
                mpiMessage(returnValue, NULL));

        if (returnValue == MPISercosMessageSERVICE_CHANNEL_ERROR) {
            MPISercosError  error;

            returnValue =
                mpiSercosError(sercos,
                               &error);

            if (returnValue == MPIMessageOK) {
```

```
                printf("\tSERCOS error 0x%x\n",
                        error);
            }
        }
    }

    /* Delete the IDN handle */
    if (idn != MPIHandleVOID) {
        deleteMessage = mpiIdnDelete(idn);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Clear Sercos Node list */
    if (sercos != MPIHandleVOID) {
        deleteMessage =
            mpiSercosNodeListSet(sercos,
                                 0,
                                 NULL);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Delete the NODE handle */
    if (node != MPIHandleVOID) {
        deleteMessage = mpiNodeDelete(node);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Delete the SERCOS handle */
    if (sercos != MPIHandleVOID) {
        deleteMessage = mpiSercosDelete(sercos);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }

    /* Delete the CONTROL handle */
    if (control != MPIHandleVOID) {
        deleteMessage = mpiControlDelete(control);

        if (returnValue == MPIMessageOK) {
            returnValue = deleteMessage;
        }
    }
```

```
    return ((int)returnValue);
}

long
    idnDisplay(MPINode          node,
               MPIIdn           idn,
               unsigned long    *attributes)
{
    MPIIdnElement   idnElement;
    MPIIdnField     field;

    long    returnValue;

    returnValue =
        mpiNodeIdnGET(node,
                      idn);

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiIdnElementGET(idn,
                             &idnElement);
    }

    if (returnValue == MPIMessageOK) {
        putchar('\n');

        for (field = MPIIdnFieldFIRST; field < MPIIdnFieldLAST; field++) {
            switch (field) {
                case MPIIdnFieldNUMBER: {
                    const char  *text;

                    if (returnValue == MPIMessageOK) {
                        returnValue =
                            mpiIdnNumberText(idn,
                                             &text);
                    }

                    if (returnValue == MPIMessageOK) {
                        printf("\tnumber %s\n",
                                text);
                    }
                    break;
                }
                case MPIIdnFieldNAME: {
                    printf("\tname %s\n",
                            idnElement.name);
                    break;
                }
                case MPIIdnFieldATTRIBUTES: {
                    printf("\tattributes 0x%x\n",
                            idnElement.attributes);
                    break;
```

```
                }
                case MPIIdnFieldUNIT: {
                    printf("\tunit %s\n",
                            idnElement.unit);
                    break;
                }
                case MPIIdnFieldMINIMUM: {
                    printf("\tminimum 0x%x\n",
                            idnElement.minimum.u);
                    break;
                }
                case MPIIdnFieldMAXIMUM: {
                    printf("\tmaximum 0x%x\n",
                            idnElement.maximum.u);
                    break;
                }
                case MPIIdnFieldDATA: {
                    if (mpiIdnFieldAttrGET(MPIIdnFieldAttrVARIABLE,
  idnElement.attributes) == 0) {
                        printf("\tdata 0x%x\n",
                                idnElement.data.binary);
                    }
                    else {
                        MPIIdnVarLength *varLength;

                        long    index;

                        varLength = &idnElement.data.varLength;

                        for (index = 0; index < varLength->count; index++) {
                            printf("\tdata[%d] ",
                                    index);

                            switch (varLength->type) {
                                case MPIIdnDataTypeBINARY: {
                                    printf("0x%x\n",
                                            varLength->as.value[index].u);
                                    break;
                                }
                                case MPIIdnDataTypeIDN: {
                                    char    idnName[32];

                                    printf("%s\n",

  mpiIdnNumberName(varLength->as.idn[index],
                                                                    idnName));
                                    break;
                                }
                                case MPIIdnDataTypeLONG: {
                                    printf("%d\n",
                                            varLength->as.value[index].l);
                                    break;
                                }
```

```
                                case MPIIdnDataTypeTEXT: {
                                    printf("%s\n",
                                            varLength->as.text);
                                    break;
                                }
                                case MPIIdnDataTypeUNSIGNED: {
                                    printf("%u\n",
                                            varLength->as.value[index].u);
                                    break;
                                }
                                case MPIIdnDataTypeUNSIGNED_HEX: {
                                    printf("0x%x\n",
                                            varLength->as.value[index].u);
                                    break;
                                }
                                default: {
                                    break;
                                }
                            }

                            if (varLength->type == MPIIdnDataTypeTEXT) {
                                break;
                            }
                        }
                    }
                    break;
                }
                default: {
                    break;
                }
            }
        }
    }

    *attributes =
        (returnValue == MPIMessageOK)
                    ? idnElement.attributes
                    : 0;

    return (returnValue);
}
```

sinit1.c -- SERCOS ring initialization

```
/* sinit1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/sinit1.c 15    7/23/01 2:36p Kevinh $";
#endif

/*

:SERCOS ring initialization

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    sinit1Main

argMainRENAME(main, sinit1)
#endif

#define BAUD_RATE_DEFAULT       (MPISercosBaud10MBIT)
#define PHASE_DEFAULT           (4)
#define SAMPLE_RATE_DEFAULT     (1000)
#define MODE_DEFAULT            (MPINodeModeOPENLOOP_POSITION_MOTOR)
#define INTENSITY_DEFAULT       (3)

/* Command line arguments and defaults */
long            baud        = -1;
long            phase       = PHASE_DEFAULT;
long            sampleRate  = SAMPLE_RATE_DEFAULT;
MPINodeMode     mode        = MODE_DEFAULT;
long            intensity   = INTENSITY_DEFAULT;
```

```
Arg argList[] = {
    {   "-baud",        ArgTypeLONG,    &baud,          },
    {   "-phase",       ArgTypeLONG,    &phase,         },
    {   "-sample",      ArgTypeLONG,    &sampleRate,    },
    {   "-mode",        ArgTypeLONG,    &mode,          },
    {   "-intensity",   ArgTypeLONG,    &intensity,     },

    {   NULL,           ArgTypeINVALID, NULL,   }
};

#if 0
#define MEI_PMC
#endif

MPISercosStatus SercosStatus;

MPIIdnList  Amplifier;
MPIIdnList  MasterData;
MPIIdnList  Phase2;
MPIIdnList  Phase3;

long
    displayError(MPISercos sercos,
                 long errorCode);

long
    idnListConfig(MPINode   node);

void
    usage(char  *programName)
{
    meiPlatformConsole("usage: %s\n\t%s\n"
                       "\t\t[-baud (%d)]\n"
                       "\t\t[-phase (%d)]\n"
                       "\t\t[-sample (%d)]\n"
                       "\t\t[-mode (%d)]\n"
                       "\t\t[-intensity (%d)]\n",
                       programName,
                       ArgUSAGE,
                       (BAUD_RATE_DEFAULT == MPISercosBaud2MBIT)  ?  2 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud4MBIT)  ?  4 :
                       (BAUD_RATE_DEFAULT == MPISercosBaud10MBIT) ? 10
                                                                  : -1,
                       PHASE_DEFAULT,
                       SAMPLE_RATE_DEFAULT,
                       MODE_DEFAULT,
                       INTENSITY_DEFAULT);

    exit(MPIMessageARG_INVALID);
}

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;
```

```
    MPISercos   sercos;

    long    returnValue = MPIMessageOK;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;
    long    sercosNumber;
    long    initialMotorNumber;

    MPISercosBaud       baudRate;
    MPISercosConfig     sercosConfig;
    MPIControlConfig    controlConfig;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        usage(argv[0]);
    }

    baudRate =
        (baud == 2)  ? MPISercosBaud2MBIT  :
        (baud == 4)  ? MPISercosBaud4MBIT  :
        (baud == 10) ? MPISercosBaud10MBIT
                     : BAUD_RATE_DEFAULT;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    returnValue = mpiControlValidate(control);

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);
```

```
    /* Configure the controller */
    returnValue =
        mpiControlConfigGet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

#if defined(MEI_PMC)
    controlConfig.axisCount     = 8;
    controlConfig.filterCount   = 8;
    controlConfig.motionCount   = 8;
    controlConfig.motorCount    = 8;
    controlConfig.sequenceCount = 8;

    controlConfig.sercosCount = 1;
#else
    controlConfig.axisCount     = 24;
    controlConfig.filterCount   = 24;
    controlConfig.motionCount   = 24;
    controlConfig.motorCount    = 24;
    controlConfig.sequenceCount = 24;

    controlConfig.sercosCount = 4;
#endif

    returnValue =
        mpiControlConfigSet(control,
                            &controlConfig,
                            NULL);
    msgCHECK(returnValue);

    initialMotorNumber = 0;

    for (sercosNumber = 0;
         sercosNumber < controlConfig.sercosCount;
         sercosNumber++) {
        MPINode         node[MPISercosNODE_COUNT_MAX];
        MPINodeConfig   nodeConfig[MPISercosNODE_COUNT_MAX];

        long    count;
        long    index;

        returnValue = MPIMessageOK;     /* reset returnValue for next sercos object
*/

        if (initialMotorNumber >= MEIXmpMAX_Motors) {
            break;
        }

        count =
            sizeof( node) /
            sizeof(*node);

        for (index = 0; index < count; index++) {
            node[index]  = MPIHandleVOID;
        }
```

```
        sercos =
            mpiSercosCreate(control,
                            sercosNumber);
        returnValue = mpiSercosValidate(sercos);

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiSercosConfigGet(sercos,
                                   &sercosConfig,
                                   NULL);
        }

        if (returnValue == MPIMessageOK) {
            sercosConfig.baudRate = baudRate;
            sercosConfig.TSCYC = sampleRate;
            sercosConfig.xmitIntensity = intensity;

            returnValue =
                mpiSercosConfigSet(sercos,
                                   &sercosConfig,
                                   NULL);
        }

        if (returnValue == MPIMessageOK) {
            returnValue =
                mpiSercosStatus(sercos,
                                &SercosStatus,
                                NULL);
        }

        if (returnValue == MPIMessageOK) {
            printf("\nsercos[%d] status: phase %d\n",
                    sercosNumber,
                    SercosStatus.phase);

            if (phase <= SercosStatus.phase) {
                goto done;
            }

            if (SercosStatus.phase < 2) {
                printf("\nsercos[%d]: transitioning to phase 2\n",
                        sercosNumber);

                if (returnValue == MPIMessageOK) {
                    returnValue =
                        mpiSercosInit(sercos,
                                      2);
                }

                if (returnValue == MPIMessageOK) {
                    returnValue =
                        mpiSercosStatus(sercos,
                                        &SercosStatus,
                                        NULL);
                }
```

```
                if (returnValue == MPIMessageOK) {
                    printf("sercos[%d] status: phase %d\n",
                            sercosNumber,
                            SercosStatus.phase);
                }
            }
        }

        /* Will be in at least Phase 2 here */
        if (returnValue == MPIMessageOK) {
            for (index = 0; index < SercosStatus.nodeCount; index++) {
                if (returnValue == MPIMessageOK) {
                    node[index] = mpiNodeCreate(sercos, index);
                    returnValue = mpiNodeValidate(node[index]);
                }

                if (returnValue == MPIMessageOK) {
                    returnValue =
                        mpiNodeConfigGet(node[index],
                                         &nodeConfig[index],
                                         NULL);
                }

                if (returnValue == MPIMessageOK) {
                    printf("\tnode[%d]: address %d\n",
                            index,
                            nodeConfig[index].address);
                }
            }
        }

        if (returnValue == MPIMessageOK) {
            if (phase > 2) {
                for (index = 0; index < SercosStatus.nodeCount; index++) {
                    if (returnValue == MPIMessageOK) {
                        nodeConfig[index].motorNumber = initialMotorNumber + index;
                        nodeConfig[index].filterNumber =
  nodeConfig[index].motorNumber;
                        nodeConfig[index].mode = mode;

                        returnValue =
                            mpiNodeConfigSet(node[index],
                                             &nodeConfig[index],
                                             NULL);
                    }

                    if(returnValue == MPIMessageOK) {
                        returnValue =
                            mpiSercosNodeAppend(sercos,
                                                node[index]);
                    }

                    if (returnValue == MPIMessageOK) {
                        returnValue =
                            idnListConfig(node[index]);
                    }
                }
```

```
                if (returnValue == MPIMessageOK) {
                    if (SercosStatus.nodeCount > 0) {
                        printf("\nsercos[%d]: transitioning to phase %d\n",
                                sercosNumber,
                                phase);

                        returnValue =
                            mpiSercosInit(sercos,
                                          phase);
                    }
                }

                if (returnValue == MPIMessageOK) {
                    returnValue =
                        mpiSercosStatus(sercos,
                                        &SercosStatus,
                                        NULL);
                }

                if (returnValue == MPIMessageOK) {
                    initialMotorNumber += SercosStatus.nodeCount;

                    printf("sercos[%d] status: phase %d\n",
                            sercosNumber,
                            SercosStatus.phase);
                }
            }
        }

done:
        if (returnValue != MPIMessageOK) {
            const char  *text;

            text = mpiMessage(returnValue, NULL);

            printf("\nSercos[%d]: Ring Initialization Error: 0x%x:%s\n",
                    sercosNumber,
                    returnValue,
                    (text == NULL)?"":text);
        }

        /* Delete the IDNLIST handles */
        if (Amplifier != MPIHandleVOID) {
            long deleteMessage = mpiIdnListDelete(Amplifier);

            Amplifier = MPIHandleVOID;

            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }
        if (MasterData != MPIHandleVOID) {
            long deleteMessage = mpiIdnListDelete(MasterData);

            MasterData = MPIHandleVOID;
```

```
            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }
        if (Phase2 != MPIHandleVOID) {
            long deleteMessage = mpiIdnListDelete(Phase2);

            Phase2 = MPIHandleVOID;

            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }
        if (Phase3 != MPIHandleVOID) {
            long deleteMessage = mpiIdnListDelete(Phase3);

            Phase3 = MPIHandleVOID;

            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }

        /* Clear Sercos Node list */
        if (sercos != MPIHandleVOID) {
            long deleteMessage =
                mpiSercosNodeListSet(sercos,
                                     0,
                                     NULL);

            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }

        /* Delete the NODE handle(s) */
        count =
            sizeof( node) /
            sizeof(*node);

        for (index = 0; index < count; index++) {
            if (node[index] != MPIHandleVOID) {
                long deleteMessage = mpiNodeDelete(node[index]);

                node[index] = MPIHandleVOID;

                if (returnValue == MPIMessageOK) {
                    returnValue = deleteMessage;
                }
            }
        }

        /* Delete the SERCOS handle */
        if (sercos != MPIHandleVOID) {
            long deleteMessage = mpiSercosDelete(sercos);

            sercos = MPIHandleVOID;
```

```
            if (returnValue == MPIMessageOK) {
                returnValue = deleteMessage;
            }
        }
    }

    /* Delete the CONTROL handle */
    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}


long
    displayError(MPISercos  sercos,
                 long       errorCode)
{
    long returnValue = MPIMessageOK;

    if (errorCode != MPIMessageOK) {
        long sercosNumber;

        returnValue =
            mpiSercosNumber(sercos,
                            &sercosNumber);

        if (returnValue == MPIMessageOK) {
            if (errorCode == MPISercosMessageSERVICE_CHANNEL_ERROR) {
                MPISercosError error;

                returnValue =
                    mpiSercosError(sercos,
                                   &error);

                if (returnValue == MPIMessageOK) {
                    printf("\nSercos[%d] Service Channel Error: 0x%x\n",
                            sercosNumber,
                            error);
                }
            }
            else {
                const char  *text;

                text = mpiMessage(errorCode, NULL);

                printf("\nSercos[%d] Error: 0x%x:%s\n",
                        sercosNumber,
                        errorCode,
                        (text == NULL)?"":text);
            }
        }
    }

    if (returnValue == MPIMessageOK) {
        returnValue = errorCode;
```

```
    }

    return (returnValue);
}

long
    idnListConfig(MPINode   node)
{
    long    returnValue = MPIMessageOK;

#if 0
    MPIIdn          idn;
#endif

#if 0
    MPIIdnData  idnData;
#endif


    /* Amplifier */
    if (returnValue == MPIMessageOK) {
        Amplifier = mpiIdnListCreate(MPIHandleVOID);
        returnValue = mpiIdnListValidate(Amplifier);
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListGet(node,
                              MPINodeIdnListTypeAMPLIFIER,
                              Amplifier);
    }

    if (returnValue == MPIMessageOK) {
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListSet(node,
                              MPINodeIdnListTypeAMPLIFIER,
                              Amplifier);
    }

    /* Master */
    if (returnValue == MPIMessageOK) {
        MasterData = mpiIdnListCreate(MPIHandleVOID);
        returnValue = mpiIdnListValidate(MasterData);
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListGet(node,
                              MPINodeIdnListTypeMASTER_DATA,
                              MasterData);
    }

    if (returnValue == MPIMessageOK) {
    }
```

```
    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListSet(node,
                              MPINodeIdnListTypeMASTER_DATA,
                              MasterData);
    }


#if 0
    /* Phase 2 */
    if (returnValue == MPIMessageOK) {
        Phase2 = mpiIdnListCreate(MPIHandleVOID);
        returnValue = mpiIdnListValidate(Phase2);
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListGet(node,
                              MPINodeIdnListTypePHASE2,
                              Phase2);
    }

    /* Set Phase 2 IDNs */
    if (returnValue == MPIMessageOK) {
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListSet(node,
                              MPINodeIdnListTypePHASE2,
                              Phase2);
    }
#endif

#if 0
    /* Phase3 */
    if (returnValue == MPIMessageOK) {
        Phase3 = mpiIdnListCreate(MPIHandleVOID);
        returnValue = mpiIdnListValidate(Phase3);
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListGet(node,
                              MPINodeIdnListTypePHASE3,
                              Phase3);
    }

    /* Set Phase 3 IDNs */
    if (returnValue == MPIMessageOK) {
    }

    if (returnValue == MPIMessageOK) {
        returnValue =
            mpiNodeIdnListSet(node,
                              MPINodeIdnListTypePHASE3,
```

```
                                Phase3);
    }
#endif
    return (returnValue);
}
```

stepcfg.c -- Configure a motor as a stepper motor

```
/* stepcfg.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/stepcfg.c 14    8/01/01 2:08p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    stepcfgMain

argMainRENAME(main, stepcfg)
#endif

/*

:Configure a motor as a stepper motor

This program will configure the motor and filter parameters for a motor to allow
 the motor to control a stepper motor.  The motor type is set to stepper, and
 the pulse width is set (default is 25.5 microseconds), and the loop back mode
 is set.  If loop back is set to TRUE, then the stepper pusles are read by the
 encoder input that is attached to the motor.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


#define MOTION_NUMBER   (0)
#define AXIS_NUMBER     (0)
#define FILTER_NUMBER   (0)
#define MOTOR_NUMBER    (0)
```

```
/* LOOPBACK_CONFIG : TRUE = enable stepper loopback, FALSE = no loopback */
#define LOOPBACK_CONFIG     TRUE
/* output pulse width  (seconds) */
#define PULSE_WIDTH_VALUE   (2.55e-5)

#undef CW_CCW_STEPPERS

#if defined CW_CCW_STEPPERS
#define IO_CONFIG_A         (MEIMotorTransceiverConfigCW)
#define IO_CONFIG_B         (MEIMotorTransceiverConfigCCW)
#else
#define IO_CONFIG_A         (MEIMotorTransceiverConfigSTEP)
#define IO_CONFIG_B         (MEIMotorTransceiverConfigDIR)
#endif

#define TRANSCEIVER_ID_A    (MEIMotorTransceiverIdA)
#define TRANSCEIVER_ID_B    (MEIMotorTransceiverIdB)
#define INVERT_BIT          (FALSE)


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                    argc,
                  char                  *argv[],
                  MPIControlType        *controlType,
                  MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress,
                 MPIMotion          *motion,
                 long                motionNumber,
                 MPIAxis            *axis,
                 long                axisNumber,
                 MPIFilter          *filter,
                 long                filterNumber,
                 MPIMotor           *motor,
                 long                motorNumber)
{
    long           returnValue;
```

```
    /* Create motion controller object */
    *control =
        mpiControlCreate(controlType,
                         controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize motion controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);

    /* Create axis object */
    *axis =
        mpiAxisCreate(*control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(*axis));

    /* Create motion supervisor object with axis */
    *motion =
        mpiMotionCreate(*control,
                        motionNumber,
                        *axis);
    msgCHECK(mpiMotionValidate(*motion));

    /* Create filter object */
    *filter =
        mpiFilterCreate(*control,
                        filterNumber);
    msgCHECK(mpiFilterValidate(*filter));

    /* Create motor object */
    *motor =
        mpiMotorCreate(*control,
                        motorNumber);
    msgCHECK(mpiMotorValidate(*motor));
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl      *control,
                    MPIMotion       *motion,
                    MPIAxis         *axis,
                    MPIFilter       *filter,
                    MPIMotor        *motor)
{
    long    returnValue;


    /* Delete motor object */
    returnValue =
        mpiMotorDelete(*motor);
    msgCHECK(returnValue);

    /* Delete filter object */
```

```
    returnValue =
        mpiFilterDelete(*filter);
    msgCHECK(returnValue);

    /* Delete motion supervisor object */
    returnValue =
        mpiMotionDelete(*motion);
    msgCHECK(returnValue);

    /* Delete axis object */
    returnValue =
        mpiAxisDelete(*axis);
    msgCHECK(returnValue);

    /* Delete motion controller object */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Disable Filter Algorithm */
void disableFilterAlgorithm(MPIFilter filter)
{
    MEIFilterConfig filterConfigXmp;
    long            returnValue;


    /* Read filter configuration */
    returnValue =
        mpiFilterConfigGet(filter,
                           NULL,
                           &filterConfigXmp);
    msgCHECK(returnValue);

    /* Set filter algorithm to NONE */
    filterConfigXmp.Algorithm = MEIXmpAlgorithmNONE;

    /* Set filter configuration */
    returnValue =
        mpiFilterConfigSet(filter,
                           NULL,
                           &filterConfigXmp);
    msgCHECK(returnValue);
}


/* Configure motor for use as a stepper motor */
void configureStepperMotor(MPIMotor                    motor,
                           MEIMotorTransceiverConfig   xcvrAType,
                           long                        xcvrAInvert,
                           MEIMotorTransceiverConfig   xcvrBType,
                           long                        xcvrBInvert,
                           float                       pulseWidth,
                           long                        loopBack)
```

```
{
    MPIMotorConfig  motorConfigMPI;
    MEIMotorConfig  motorConfigXMP;
    long            returnValue;


    /* Read motor configuration */
    returnValue =
        mpiMotorConfigGet(motor,
                          &motorConfigMPI,
                          &motorConfigXMP);
    msgCHECK(returnValue);

    /* Setup motor as a stepper motor */
    motorConfigMPI.type = (MPIMotorType)MEIXmpMotorTypeSTEPPER;

    motorConfigMPI.event[MPIEventTypeAMP_FAULT].action   = MPIActionNONE;
    motorConfigMPI.event[MPIEventTypeLIMIT_ERROR].action = MPIActionNONE;

    motorConfigXMP.Transceiver[TRANSCEIVER_ID_A].Config = xcvrAType;
    motorConfigXMP.Transceiver[TRANSCEIVER_ID_B].Config = xcvrBType;
    motorConfigXMP.Transceiver[TRANSCEIVER_ID_A].Invert = xcvrAInvert;
    motorConfigXMP.Transceiver[TRANSCEIVER_ID_B].Invert = xcvrBInvert;

    motorConfigXMP.Stepper.PulseWidth = pulseWidth;

    motorConfigXMP.Stepper.Loopback = loopBack;

    /* Set motor configuration */
    returnValue =
        mpiMotorConfigSet(motor,
                          &motorConfigMPI,
                          &motorConfigXMP);
    msgCHECK(returnValue);
}


/* Set in position fine tolerance */
void setAxisFineTolerance(MPIAxis   axis,
                          float     fineTolerance)
{
    MPIAxisConfig   axisConfig;
    long            returnValue;


    /* Read axis configuration */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    /* Set axis in position fine tolerance */
    axisConfig.inPosition.tolerance.positionFine = fineTolerance;
```

```
    /* Set axis configuration */
    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);
}

int main(int    argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;
    MPIMotion           motion;
    MPIAxis             axis;
    MPIFilter           filter;
    MPIMotor            motor;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress,
                &motion,
                MOTION_NUMBER,
                &axis,
                AXIS_NUMBER,
                &filter,
                FILTER_NUMBER,
                &motor,
                MOTOR_NUMBER);

    /* Disable Filter Algorithm */
    disableFilterAlgorithm(filter);

    /* Configure motor for use as a stepper motor */
    configureStepperMotor(motor,
                          IO_CONFIG_A,
                          INVERT_BIT,
                          IO_CONFIG_B,
                          INVERT_BIT,
                          (float)PULSE_WIDTH_VALUE,
                          LOOPBACK_CONFIG);

    /* Set in position fine tolerance */
    setAxisFineTolerance(axis,
                         (float)1.E8);
```

```
    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control,
                   &motion,
                   &axis,
                   &filter,
                   &motor);

    return MPIMessageOK;
}
```

stepcfg2.c -- Configure an 8 axis XMP to support 8 Servos and 8 Steppers

```
/* stepCfg2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/stepcfg2.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Configure an 8 axis XMP to support 8 Servos and 8 Steppers

The XMP controller has the capability to control 8 servo axes as well
 as 8 stepper axes on a single controller.  The XMP uses the Xcvr
 IO for axes 0 - 7 for stepper outputs on axes 8 - 15.  This is obtained
 by redirecting the stepper controller pointer from Xcvr hardware that
 would be present on a 16 axis controller to Xcvr's 0 - 7.

Certain limitations apply:
 -No Steppers can be configured on the first 8 axes.
 -The 2nd 8 axes will not have any IO (User, Home, PosLimits, AmpEnables, etc).
 -No encoder loopback is availiable on the Stepper axes.
 -The IN_FINE position criteria must be increased for motion done to return.

The following code performs the necessary configurations to run a Step/Dir
 stepper on Motor 8 using Xcvr0 A & B.  Define CONFIG_ONLY to configure the
 XMP, but not perform any motion.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/



#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    stepCfg2Main
```

```
argMainRENAME(main, stepCfg2)
#endif

#define AXIS_COUNT      (16)
#define FILTER_COUNT    (16)
#define MS_COUNT        (16)
#define MOTOR_COUNT     (16)

/* Command line arguments and defaults */
long    axisNumber[AXIS_COUNT]      = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,};
long    filterNumber[FILTER_COUNT]  = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,};
long    motorNumber[MOTOR_COUNT]    = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,};
long    motionNumber                = -1;   /* Use next availiable MS */
long    width                       = 255;

Arg argList[] = {
    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* LOOPBACK_CONFIG : TRUE = count step pulses in encoder register, FALSE = no
loopback */
#define LOOPBACK_CONFIG     TRUE
/* output pulse width  (seconds) */
#define PULSE_WIDTH_VALUE   (2.55e-5)

#undef CW_CCW_STEPPERS

#if defined CW_CCW_STEPPERS
#define IO_CONFIG_A         (MEIMotorTransceiverConfigCW)
#define IO_CONFIG_B         (MEIMotorTransceiverConfigCCW)
#else
#define IO_CONFIG_A         (MEIMotorTransceiverConfigSTEP)
#define IO_CONFIG_B         (MEIMotorTransceiverConfigDIR)
#endif

#define TRANSCEIVER_ID_A    (MEIXmpMotorIOBitXCVR_A)
#define TRANSCEIVER_ID_B    (MEIXmpMotorIOBitXCVR_B)
#define INVERT_BIT          (FALSE)

int
    main(int    argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlConfig    config;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    MPIFilter           filter;

    MPIAxis             axis;
    MPIAxisConfig       axisConfig;     /* Axis configuration */

    MPIMotion           motion;

    MPIMotor            motorZero;
```

```
    MPIMotor            motorEight;
    MPIMotorConfig      motorZeroConfig;
    MPIMotorConfig      motorEightConfig;

    MEIMotorConfig      motorZeroConfigXmp;
    MEIMotorConfig      motorEightConfigXmp;

    MPIMotionParams params;     /* motion parameters */
    MPITrajectory   trajectory; /* motion trajectory */
    double          position;   /* final target position */

    long    argIndex;
    long    returnValue;
    long    motionDone = FALSE;
    long    stepperNumber;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc)) {
        meiPlatformConsole("usage: %s %s\n",
                           argv[0],
                           ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }

    /* Obtain a Control handle */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize the controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Get controller status */
```

```
    returnValue =
        mpiControlConfigGet(control,
                            &config,
                            NULL);
    msgCHECK(returnValue);

    /* Configure controller for 16 Axes, Filters, and Motors */
    config.axisCount            = AXIS_COUNT;
    config.filterCount          = FILTER_COUNT;
    config.motionCount          = MS_COUNT;
    config.motorCount           = MOTOR_COUNT;

    /* Set controller configuration */
    returnValue =
        mpiControlConfigSet(control,
                            &config,
                            NULL);
    msgCHECK(returnValue);

    /* Display controller configuration */
    printf("Controller:\n"
            "\taxisCount %d\n"
            "\tadcCount  %d\n"
            "\t cmdDacCount  %d\n"
            "\tfilterCount %d\n"
            "\tmotionCount %d\n"
            "\tmotorCount %d\n"
            "\tsequenceCount %d\n"
            "\tsampleRate    %d\n",
            config.axisCount,
            config.adcCount,
            config.cmdDacCount,
            config.filterCount,
            config.motionCount,
            config.motorCount,
            config.sequenceCount,
            config.sampleRate);

    /* Create filter object */
    filter =
        mpiFilterCreate(control,
                        filterNumber[8]);
    msgCHECK(mpiFilterValidate(filter));

    /* Configure Filter for Step algorithm */
    if (returnValue == MPIMessageOK) {
        MEIFilterConfig filterConfigXmp;

        returnValue =
            mpiFilterConfigGet(filter,
                               NULL,
                               &filterConfigXmp);
        msgCHECK(returnValue);

        filterConfigXmp.Algorithm = MEIXmpAlgorithmNONE;

        returnValue =
```

```
            mpiFilterConfigSet(filter,
                               NULL,
                               &filterConfigXmp);
        msgCHECK(returnValue);

    }

    /* Create axis object*/
    axis =
        mpiAxisCreate(control,
                      axisNumber[8]);       /* axis number */
    msgCHECK(mpiAxisValidate(axis));

    /* Configure axis */
    returnValue =
        mpiAxisConfigGet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    axisConfig.inPosition.tolerance.positionFine = (float)10E10;

    returnValue =
        mpiAxisConfigSet(axis,
                         &axisConfig,
                         NULL);
    msgCHECK(returnValue);

    /* Create motion object, append axis */
    motion =
        mpiMotionCreate(control,
                        motionNumber,   /* motion supervisor number */
                        axis);          /* axis object handle */
    msgCHECK(mpiMotionValidate(motion));

    /* Set up motion parameters */
    trajectory.velocity     =  1000.0;      /* counts per sec */
    trajectory.acceleration = 10000.0;      /* counts per sec * sec */
    trajectory.deceleration = 10000.0;

    position = 2000.0;          /* target position (counts) */

    params.trapezoidal.trajectory   = &trajectory;
    params.trapezoidal.position     = &position;

    /* Create Motor0 Object */
    motorZero =
        mpiMotorCreate(control,
                       0);
    msgCHECK(mpiMotorValidate(motorZero));

    /* Create Motor8 Object */
    motorEight =
        mpiMotorCreate(control,
                       8);
        msgCHECK(mpiMotorValidate(motorEight));
```

```
    /* Get Motor0's Configuration */
    returnValue =
        mpiMotorConfigGet(motorZero,
                          &motorZeroConfig,
                          &motorZeroConfigXmp);
    msgCHECK(returnValue);

    /* Configure Xcvr A/B for Step/Dir */
    motorZeroConfigXmp.Transceiver[TRANSCEIVER_ID_A].Config = IO_CONFIG_A;
    motorZeroConfigXmp.Transceiver[TRANSCEIVER_ID_B].Config = IO_CONFIG_B;
    motorZeroConfigXmp.Transceiver[TRANSCEIVER_ID_A].Invert = INVERT_BIT;
    motorZeroConfigXmp.Transceiver[TRANSCEIVER_ID_B].Invert = INVERT_BIT;

    returnValue =
        mpiMotorConfigSet(motorZero,
                           &motorZeroConfig,
                           &motorZeroConfigXmp);
    msgCHECK(returnValue);

    printf("StepControl Pointer for Motor 0: 0x%X\n",
motorZeroConfigXmp.Commutation.StepControl);

    /* Get MotorEight's Configuration */
    returnValue =
        mpiMotorConfigGet(motorEight,
                          &motorEightConfig,
                          &motorEightConfigXmp);
    msgCHECK(returnValue);

    motorEightConfigXmp.Stepper.PulseWidth = (float)PULSE_WIDTH_VALUE;
    motorEightConfigXmp.Stepper.Loopback = LOOPBACK_CONFIG;

    /* Set motorEight's Commutation.StepControl register equal to MotorZero's */
    stepperNumber = motorNumber[0];

    returnValue =
        meiMotorConfigStepper(motorEight,
                              &motorEightConfigXmp,
                              stepperNumber);  /* StepControl source location */
    msgCHECK(returnValue);

    /*  Disable limit error */
    motorEightConfig.event[MPIEventTypeLIMIT_ERROR].action          = MPIActionNONE;
    motorEightConfig.event[MPIEventTypeLIMIT_ERROR].trigger.error   = (float)10E10;

    /*  Disable home event  */
    motorEightConfig.event[MPIEventTypeHOME].action         = MPIActionNONE;

    /*  Disable amp fault event */
    motorEightConfig.event[MPIEventTypeAMP_FAULT].action    = MPIActionNONE;

    /*  Disable neg. HW limit event */
    motorEightConfig.event[MPIEventTypeLIMIT_HW_NEG].action = MPIActionNONE;

    /*  Disable pos. HW limit event */
    motorEightConfig.event[MPIEventTypeLIMIT_HW_POS].action = MPIActionNONE;
```

```
    /*  Disable neg. SW limit event */
    motorEightConfig.event[MPIEventTypeLIMIT_SW_NEG].action = MPIActionNONE;

    /*  Disable pos. SW limit event */
    motorEightConfig.event[MPIEventTypeLIMIT_SW_POS].action = MPIActionNONE;

    /*  Disable Encoder Fault event */
    motorEightConfig.event[MPIEventTypeENCODER_FAULT].action= MPIActionNONE;

    /*  Configure Motor 8 as Stepper */
    motorEightConfig.type = MPIMotorTypeSTEPPER;

    returnValue =
        mpiMotorConfigSet(motorEight,
                          &motorEightConfig,
                          &motorEightConfigXmp);
    msgCHECK(returnValue);

    /* Write MS/Axis map to controller and clear any events */
    returnValue =
        mpiMotionAction(motion,
                        MPIActionRESET);
    msgCHECK(mpiMotionValidate(motion));

    /* Verify it worked */
    returnValue =
        mpiMotorConfigGet(motorEight,
                          &motorEightConfig,
                          &motorEightConfigXmp);
    msgCHECK(returnValue);

    printf("New StepControl Pointer for Motor 8: 0x%X\n",
motorEightConfigXmp.Commutation.StepControl);

#ifndef CONFIG_ONLY

    printf("Motor 8 configured for Stepper using Xcvr 0 & 1.\n"
           "Starting move on Motor 8...\n");

    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
                       &params);
    msgCHECK(returnValue);

    /* Poll status until motion done */
    motionDone = FALSE;
    while (motionDone == FALSE) {
        MPIStatus   status;

        returnValue =
            mpiMotionStatus(motion,
                            &status,
                            NULL);
        msgCHECK(returnValue);

        switch (status.state) {
```

```
            case MPIStateIDLE: {
                motionDone = TRUE;

                /* Wait for the motor to settle */
                meiPlatformSleep(300);  /* msec */

                break;
            }

            case MPIStateERROR: {
                motionDone = TRUE;

                /* Clear any error condition(s) */
                returnValue =
                    mpiMotionAction(motion,
                                    MPIActionRESET);
                msgCHECK(returnValue);

                /* Wait for reset to take effect */
                meiPlatformSleep(100);      /* msec */

                break;
            }
            default: {
                break;
            }
        }
    }

    position = 0.0;         /* target position (counts) */
    params.trapezoidal.position     = &position;

    returnValue =
        mpiMotionStart(motion,
                       MPIMotionTypeTRAPEZOIDAL,
                       &params);
    msgCHECK(returnValue);

#endif /* CONFIG_ONLY */

    /* Delete object handles */
    returnValue = mpiMotorDelete(motorZero);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motorEight);
    msgCHECK(returnValue);

    returnValue = mpiFilterDelete(filter);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

stoprate.c -- Set deceleration rate for MPIActionSTOP and MPIActionE_STOP.

```
/* stoprate.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/stoprate.c 8     7/23/01 2:36p Kevinh $";
#endif

/*

:Set deceleration rate for MPIActionSTOP and MPIActionE_STOP.

This sample code demonstrates how to configure the MPIActionSTOP and
MPIActionE_STOP deceleration rates for a given Motion objectr.  Unless
specified at the command line, the DEFAULT deceleration rates will be used.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    stoprateMain

argMainRENAME(main, stoprate)
#endif

/* Define STOP and ESTOP rates in seconds */
#define STOP_RATE_DEFAULT   ((float)3.0)
#define ESTOP_RATE_DEFAULT  ((float)1.0)

/* Command line arguments and defaults */
long    motionNumber    = 0;
float   stopRate        = STOP_RATE_DEFAULT;
```

```
float   eStopRate       = ESTOP_RATE_DEFAULT;

Arg argList[] = {
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-stop",    ArgTypeFLOAT,   &stopRate,      },
    {   "-estop",   ArgTypeFLOAT,   &eStopRate,     },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

int
    main(int    argc,
         char   *argv[])
{

    MPIControl  control;        /* Motion controller handle */
    MPIMotion   motion;         /* Motion supervisor handle */

    MPIMotionConfig     motionConfig;   /* Motion supervisor configuration */

    long    returnValue;            /* Return value from library */

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (motionNumber >= MEIXmpMAX_MSs)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-stop (%f)]\n"
                           "\t\t[-estop (%f)]\n",
```

```
                        argv[0],
                        ArgUSAGE,
                        MEIXmpMAX_MSs - 1,
                        STOP_RATE_DEFAULT,
                        ESTOP_RATE_DEFAULT);
    exit(MPIMessageARG_INVALID);
}

/* Create controller object */
control =
    mpiControlCreate(controlType,
                     &controlAddress);
msgCHECK(mpiControlValidate(control));

/* Initialize motion controller */
returnValue = mpiControlInit(control);
msgCHECK(returnValue);

/* Create motion object */
motion =
    mpiMotionCreate(control,
                    motionNumber,
                    MPIHandleVOID);
msgCHECK(mpiMotionValidate(motion));

/* Read current motion supervisor configuration */
returnValue =
    mpiMotionConfigGet(motion,
                       &motionConfig,
                       NULL);
msgCHECK(returnValue);

printf("OLD: motion %d: stopRate %f eStopRate %f\n",
        motionNumber,
        motionConfig.decelTime.stop,
        motionConfig.decelTime.eStop);

/* Set new STOP and E_STOP deceleration rates */
motionConfig.decelTime.stop  = stopRate;
motionConfig.decelTime.eStop = eStopRate;

returnValue =
    mpiMotionConfigSet(motion,
                       &motionConfig,
                       NULL);
msgCHECK(returnValue);

printf("NEW: motion %d: stopRate %f eStopRate %f\n",
        motionNumber,
        stopRate,
        eStopRate);

/* Delete objects */

returnValue = mpiMotionDelete(motion);
```

```
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}
```

template.c -- Template application (one line description for sample app)

```
/* template.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
    "$Header: /MainTree/XMPLib/XMP/app/template.c 13    7/18/01 2:45p Kevinh $";
#endif

#if defined(ARG_MAIN_RENAME)
#define main    templateMain
argMainRENAME(main, template)
#endif

/*

:Template application (one line description for sample app)

detailed description

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.

The msgCHECK(...) macros used in the following sample code are intended
 to convey our strong belief that ALL error return codes should be checked.
 Actual application code should use specific error handling techniques (other
 than msgCHECKs) best suited to your internal error recovery methods.

*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"


/* Perform basic command line parsing. (-control -server -port -trace) */
void basicParsing(int                   argc,
                  char                 *argv[],
                  MPIControlType       *controlType,
```

```
                    MPIControlAddress     *controlAddress)
{
    long argIndex;

    /* Parse command line for Control type and address */
    argIndex = argControl(argc, argv, controlType, controlAddress);

    /* Check for unknown/invalid command line arguments */
    if (argIndex < argc) {
        fprintf(stderr,"usage: %s %s\n", argv[0], ArgUSAGE);
        exit(MPIMessageARG_INVALID);
    }
}


/* Create and initialize MPI objects */
void programInit(MPIControl         *control,
                 MPIControlType      controlType,
                 MPIControlAddress  *controlAddress)
{
    long    returnValue;


    /* Obtain a control handle */
    *control =
        mpiControlCreate(controlType, controlAddress);
    msgCHECK(mpiControlValidate(*control));

    /* Initialize the controller */
    returnValue =
        mpiControlInit(*control);
    msgCHECK(returnValue);
}


/* Perform certain cleanup actions and delete MPI objects */
void programCleanup(MPIControl *control)
{
    long    returnValue;

    /* Delete control handle */
    returnValue =
        mpiControlDelete(*control);
    msgCHECK(returnValue);
}


/* Your first function */
void function1()
{
    printf(""); /* Dummy function call - Replace with useful code */
}


/* Your second function */
```

```
void function2()
{
    printf(""); /* Dummy function call - Replace with useful code */
}


int main(int     argc,
         char   *argv[])
{
    MPIControl          control;
    MPIControlType      controlType;
    MPIControlAddress   controlAddress;


    /* Perform basic command line parsing. (-control -server -port -trace) */
    basicParsing(argc,
                 argv,
                 &controlType,
                 &controlAddress);

    /* Create and initialize MPI objects */
    programInit(&control,
                controlType,
                &controlAddress);

    /* Place your code here! */
    function1();
    function2();

    /* Perform certain cleanup actions and delete MPI objects */
    programCleanup(&control);

    return MPIMessageOK;
}
```

---

usrlim1.c -- User Limit to watch an input bit and set an output bit.

```
/* UsrLim1.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

#if defined(MEI_RCS)
static const char MEIAppRCS[] =
"$Header: /MainTree/XMPLib/XMP/app/usrlim1.c 12    7/23/01 2:36p Kevinh $";
#endif

/*

:User Limit to watch an input bit and set an output bit.

This sample code shows how to configure the XMP controller's User Limits
 to compare an input bit to a specific pattern.  If the pattern matches,
 then the specified output bit is activated and a User Event is generated
 to the host.


The XMP-Series controller "User Limit" feature allows the user to program a
 the result of two logical condition to generate an event to the host.  Also,
 a "User Limit" can be configured to write an "output" to any XMP memory
 location, using an AND mask and OR mask.

Here is the "User Limit" block structure:

typedef struct {
    MEIXmpLimitType         Type;
    void                    *SourceAddress;
    long                    Mask;
    MEIXmpGenericValue      LimitValue;
} MEIXmpLimitCondition;

typedef struct {
    long                    AndMask;
    long                    OrMask;
    long                    *OutputPtr;
    long                    Enabled;
} MEIXmpLimitOutput;

typedef struct {
    MEIXmpLimitCondition    Condition[MEIXmpLimitConditions];
    MEIXmpStatus            Status;
    MEIXmpLogic             Logic;
    MEIXmpLimitOutput       Output;
    long                    Count;
    long                    State;
```

```
} MEIXmpLimitData;


MEIXmpLimitTypes are the operators that are used for the User Limit's
 Condition[0] and Condition[1].  They are found in xmp.h.

*SourceAddress is a pointer to an XMP memory location.

Mask is ANDed with the value located at the *SourceAddress.

LimitValue is compared with the masked value located at *SourceAddress,
 using the Type operator.

Status defines the status bit used to generate an event to the host.

MEIXmpLogic is the logic applied between the two condition block outputs,
 Condition[0] and Condition[1]:

 MEIXmpLogicNEVER
    Does NOT evaluate Condition[0], Condition[1].  No event is generated.

 MEIXmpLogicSINGLE
    Only evaluates Condition[0].  Event is generated if Condition[0] is TRUE.

 MEIXmpLogicOR
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    OR Condition[1]) = TRUE.

 MEIXmpLogicAND
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    AND Condition[1]) = TRUE.

The other MEIXmpLogic enums in xmp.h are for internal use only.

*OutputPtr is a pointer to an XMP memory location.

AndMask is ANDed, and OrMask is ORed with the value located at
 *OutputPtr when the resultant MEIXmpLogic applied Condition[0] and
 Condition[1] are TRUE.

Count and State are for internal use only.  Do not write these values.
  The MPI method, mpiMotorEventConfigSet(...) will not write these values.


The XMP supports up to 8 User Limits per motor.  The User Limit processing
 occurs in the firmware background task.  For maximum efficiency, the XMP only
 processes the User Limits (in order 0, 1, 2, etc.), up to the largest User
 Limit number that has (motorEventConfig.Logic ! = MEIXmpLogicNEVER).

For example:
 If UserLimit 0 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    only process the first UserLimit for each motor.
 If UserLimit 3 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    process UserLimits number 0, 1, 2, and 3 for each motor.

It's best to use the lower numbered UserLimits for maximum efficiency.  When
 finished using UserLimits, it's a good idea to set the Logic to
```

```
 MEIXmpLogicNEVER.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    UsrLim1Main

argMainRENAME(main, UsrLim1)
#endif


/* Motor I/O Configurations */
#define INPUT_BIT       (MEIMotorTransceiverIdC)    /* A, B, or C */
#define INPUT_MASK      (MEIMotorTransceiverMaskC)  /* mask for input bit */
#define ACTIVE_STATE    (FALSE)                     /* trigger on active or inactive
*/

#define OUTPUT_BIT      (MEIMotorTransceiverIdB)    /* A, B, or C */
#define OUTPUT_MASK     (MEIMotorTransceiverMaskB)  /* mask for output bit */
#define OUTPUT_INVERT   (TRUE)                      /* inverted or normal */

/* User Limit Event configuration */
#define EVENT_TYPE      (MEIEventTypeLIMIT_USER0)   /* 0, 1, 2...7 */

/* Notify Wait Timeout */
#define CYCLE_TIME          (100)                   /* msec */

/* Command line arguments and defaults */
long            motorNumber     = 0;

Arg argList[] = {
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

/* Function Prototypes */
long XcvrUserLimitSet(MPIMotor motor,
                      long inputMask,
                      long inputActiveState,
                      long outputMask,
                      MEIEventType eventType);


int
    main(int    argc,
```

```
          char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPIMotor    motor;      /* motor object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    MEIMotorConfig  motorConfigXmp;  /* XMP motor I/O configuration */

    long    motionNumber;   /* motion supervisor number */
    long    axisNumber;
    long    returnValue;    /* return value from library */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (motorNumber >= MEIXmpMAX_Motors)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-motor # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1);
        exit(MPIMessageARG_INVALID);
```

```
    }

    /* Use a one to one relationship between motor, axis, and motion */
    motionNumber = motorNumber;
    axisNumber = motorNumber;

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create motor object for motorNumber */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create axis object for axisNumber */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motion object, appending the axis object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Request notification of ALL events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);

    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
```

```
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Configure the specified INPUT_BIT */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Transceiver[INPUT_BIT].Config = MEIMotorTransceiverConfigINPUT;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Configure the specified OUTPUT_BIT */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Transceiver[OUTPUT_BIT].Config = MEIMotorTransceiverConfigOUTPUT;
    motorConfigXmp.Transceiver[OUTPUT_BIT].Invert = OUTPUT_INVERT;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Configure User Limit */
    returnValue =
        XcvrUserLimitSet(motor,
                         INPUT_MASK,
                         ACTIVE_STATE,
                         OUTPUT_MASK,
                         EVENT_TYPE);
    msgCHECK(returnValue);

    printf("Press any key to exit ...\n");

    /* Loop repeatedly */
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus      eventStatus;
```

```
        MEIEventStatusInfo  *info;

        /* Wait for events */
        returnValue =
            mpiNotifyEventWait(notify,
                               &eventStatus,
                               (MPIWait)CYCLE_TIME);

        if (returnValue == MPIMessageTIMEOUT) {     /* ignore timeouts */
            returnValue = MPIMessageOK;
        }
        else {
            msgCHECK(returnValue);

            info = (MEIEventStatusInfo *)eventStatus.info;

            switch (eventStatus.type) {
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER0: {
                    printf("\nUser Limit #0, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER1: {
                    printf("\nUser Limit #1, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER2: {
                    printf("\nUser Limit #2, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER3: {
                    printf("\nUser Limit #3, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER4: {
                    printf("\nUser Limit #4, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
```

```
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER5: {
                    printf("\nUser Limit #5, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER6: {
                    printf("\nUser Limit #6, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }
                /* User Limit Event from motor source */
                case MEIEventTypeLIMIT_USER7: {
                    printf("\nUser Limit #7, type %d source 0x%x info 0x%x",
                            eventStatus.type,
                            eventStatus.source,
                            eventStatus.info[0]);
                    break;
                }

                default: {
                    printf("mpiNotifyEventWait() returns 0x%x\n"
                        "\teventStatus: type %d source 0x%x info 0x%x\n",
                        returnValue,
                        eventStatus.type,
                        eventStatus.source,
                        eventStatus.info[0]);
                    break;
                }
            }
        }
    }

    printf("\n");

    /* Delete objects */
    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = mpiMotorDelete(motor);
```

```
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}


long XcvrUserLimitSet(MPIMotor motor,
                      long inputMask,
                      long inputActiveState,
                      long outputMask,
                      MEIEventType eventType)
{
    MPIControl          control;

    MEIXmpData          *firmware;


    MEIMotorEventConfig motorEventConfig;
    MEIXmpStatus        status;

    long                returnValue;
    long                motorIndex;
    MPIEventMask        resetMask;

    /* Determine motor number */
    returnValue =
        mpiMotorNumber(motor,
                       &motorIndex);

    /* Determine control handle */
    if(returnValue == MPIMessageOK) {
        control = mpiMotorControl(motor);
        returnValue = mpiControlValidate(control);
    }

    /* Get pointer to XMP firmware */
    if(returnValue == MPIMessageOK) {
        returnValue =
            mpiControlMemory(control,
                             (void **)&firmware,
                             NULL);
    }

    mpiEventMaskCLEAR(resetMask);

    if (returnValue == MPIMessageOK) {
        switch (eventType) {
            case MEIEventTypeLIMIT_USER0: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER0);
                break;
            }
            case MEIEventTypeLIMIT_USER1: {
                status = MEIXmpStatusLIMIT;
```

```
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER1);
                break;
            }
            case MEIEventTypeLIMIT_USER2: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER2);
                break;
            }
            case MEIEventTypeLIMIT_USER3: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER3);
                break;
            }
            case MEIEventTypeLIMIT_USER4: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER4);
                break;
            }
            case MEIEventTypeLIMIT_USER5: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER5);
                break;
            }
            case MEIEventTypeLIMIT_USER6: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER6);
                break;
            }
            case MEIEventTypeLIMIT_USER7: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER7);
                break;
            }
            default:
                printf("\nError, invalid event type\n");
                break;
        }

        returnValue =
            mpiMotorEventConfigGet(motor,
                                   (MPIEventType)eventType,
                                   NULL,
                                   &motorEventConfig);
    }

    if (returnValue == MPIMessageOK) {
        /* Set up the Condition[0] block.  The operator between the value in
            the SourceAddress (masked by the Mask), and LimitValue.l, is
            MEIXmpLimitTypeBIT_CMP (equals).  Thus, when the value in
            SourceAddress equals the value in LimitValue.l, then the
            Condition[0] block is TRUE.
        */
        motorEventConfig.Condition[0].Type = MEIXmpLimitTypeBIT_CMP;
        motorEventConfig.Condition[0].SourceAddress =
            &firmware->Motor[motorIndex].IO.DedicatedIN.IO;     /* input address */
        motorEventConfig.Condition[0].Mask = inputMask;         /* AND mask */
        motorEventConfig.Condition[0].LimitValue.l =
```

```
            inputActiveState ? inputMask : 0x0;                 /* expected value */

        /* Set up the Condition[1] block.  The operator is MEIXmpLimitTypeFALSE.
            Therefore, the Condition[1] block is always FALSE.
        */
        motorEventConfig.Condition[1].Type = MEIXmpLimitTypeFALSE;
        motorEventConfig.Condition[1].SourceAddress = NULL;
        motorEventConfig.Condition[1].Mask = 0;
        motorEventConfig.Condition[1].LimitValue.l = 0;

        motorEventConfig.Status = status;
        /* Determine logic result from Condition[0] only */
        motorEventConfig.Logic  = MEIXmpLogicSINGLE;

        motorEventConfig.Output.OutputPtr =
            &firmware->Motor[motorIndex].IO.MotorOutput;    /* output address */
        motorEventConfig.Output.AndMask = 0xffffffff;
        motorEventConfig.Output.OrMask = outputMask;
        motorEventConfig.Output.Enabled = TRUE;

        returnValue =
            mpiMotorEventConfigSet(motor,
                                   (MPIEventType)eventType,
                                   NULL,
                                   &motorEventConfig);
    }

    /* Reset event in case conditions have already been satisfied */
    if (returnValue == MPIMessageOK) {
        returnValue =
                mpiMotorEventReset(motor,
                                   resetMask);
    }

    return returnValue;
}
```

usrlim2.c -- Position comparison using User Limits to generate events.

```
/* UsrLim2.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Position comparison using User Limits to generate events.

This sample code shows how to configure the XMP controller's User Limits
 to compare an axis' actual position to a specific position value.  If the
 actual position is greater than or equal to the value specified, then
 the controller will generate a User Event to the Host.

The function CompareLimitSet(...) demonstrates how to configure a
 User Limit for position compare.


The XMP-Series controller "User Limit" feature allows the user to program a
 the result of two logical condition to generate an event to the host.  Also,
 a "User Limit" can be configured to write an "output" to any XMP memory
 location, using an AND mask and OR mask.

Here is the "User Limit" block structure:

typedef struct {
    MEIXmpLimitType         Type;
    void                    *SourceAddress;
    long                    Mask;
    MEIXmpGenericValue      LimitValue;
} MEIXmpLimitCondition;

typedef struct {
    long                    AndMask;
    long                    OrMask;
    long                    *OutputPtr;
    long                    Enabled;
} MEIXmpLimitOutput;

typedef struct {
    MEIXmpLimitCondition    Condition[MEIXmpLimitConditions];
    MEIXmpStatus            Status;
    MEIXmpLogic             Logic;
    MEIXmpLimitOutput       Output;
    long                    Count;
    long                    State;
```

```
} MEIXmpLimitData;


MEIXmpLimitTypes are the operators that are used for the User Limit's
 Condition[0] and Condition[1].  They are found in xmp.h.

*SourceAddress is a pointer to an XMP memory location.

Mask is ANDed with the value located at the *SourceAddress.

LimitValue is compared with the masked value located at *SourceAddress,
 using the Type operator.

Status defines the status bit used to generate an event to the host.

MEIXmpLogic is the logic applied between the two condition block outputs,
 Condition[0] and Condition[1]:

 MEIXmpLogicNEVER
    Does NOT evaluate Condition[0], Condition[1].  No event is generated.

 MEIXmpLogicSINGLE
    Only evaluates Condition[0].  Event is generated if Condition[0] is TRUE.

 MEIXmpLogicOR
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    OR Condition[1]) = TRUE.

 MEIXmpLogicAND
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    AND Condition[1]) = TRUE.

The other MEIXmpLogic enums in xmp.h are for internal use only.

*OutputPtr is a pointer to an XMP memory location.

AndMask is ANDed, and OrMask is ORed with the value located at
 *OutputPtr when the resultant MEIXmpLogic applied Condition[0] and
 Condition[1] are TRUE.

Count and State are for internal use only.  Do not write these values.
  The MPI method, mpiMotorEventConfigSet(...) will not write these values.


The XMP supports up to 8 User Limits per motor.  The User Limit processing
 occurs in the firmware background task.  For maximum efficiency, the XMP only
 processes the User Limits (in order 0, 1, 2, etc.), up to the largest User
 Limit number that has (motorEventConfig.Logic ! = MEIXmpLogicNEVER).

For example:
 If UserLimit 0 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    only process the first UserLimit for each motor.
 If UserLimit 3 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    process UserLimits number 0, 1, 2, and 3 for each motor.
```

```
 It's best to use the lower numbered UserLimits for maximum efficiency.  When
  finished using UserLimits, it's a good idea to set the Logic to
  MEIXmpLogicNEVER.

 Warning!  This is a sample program to assist in the integration of the
  XMP motion controller with your application.  It may not contain all
  of the logic and safety features that your application requires.
 */

 #include <stdlib.h>
 #include <stdio.h>

 #include "stdmpi.h"
 #include "stdmei.h"

 #include "apputil.h"

 #if defined(ARG_MAIN_RENAME)
 #define main    UsrLim2Main

 argMainRENAME(main, UsrLim2)
 #endif

 #define MOTION_COUNT        (2)

 /* User Limit configurations */
 #define EVENT_TYPE          (MEIEventTypeLIMIT_USER0)   /* 0, 1, 2...7 */
 #define COMPARE_POSITION    (10000)                     /* Actual Position */

 /* Command line arguments and defaults */
 long            axisNumber      = 0;
 long            motionNumber    = 0;
 long            motorNumber     = 0;
 MPIMotionType   motionType      = MPIMotionTypeS_CURVE;

 Arg argList[] = {
     {   "-axis",    ArgTypeLONG,    &axisNumber,    },
     {   "-motion",  ArgTypeLONG,    &motionNumber,  },
     {   "-motor",   ArgTypeLONG,    &motorNumber,   },
     {   "-type",    ArgTypeLONG,    &motionType,    },

     {   NULL,       ArgTypeINVALID, NULL,   }
 };

 double position[MOTION_COUNT] = {
     20000.0,
     0.0,
 };

 MPITrajectory trajectory[MOTION_COUNT] = {
     /* velocity     accel       decel       jerkPercent */
     { 10000.0,      1000000.0,  1000000.0,  0.0,    },
     { 10000.0,      100000.0,   100000.0,   0.0,    },
 };
```

```
/* Motion Parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0], &position[0],   },
    {   &trajectory[1], &position[1],   },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0], &position[0],   },
    {   &trajectory[1], &position[1],   },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0], },
    {   &trajectory[1], },
};


/* Function Prototypes */
long CompareLimitSet(MPIAxis axis,
                     MPIMotor motor,
                     long position,
                     MEIEventType eventType);


int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPIMotor    motor;      /* motor object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;

    long    returnValue;    /* return value from library */

    long    index;
    long    motionDone;     /* flag when Done occurs */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);
```

```
    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motorNumber >= MEIXmpMAX_Motors) ||
        (motionType < MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
```

```
                        &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object for axisNumber */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motor object for motorNumber */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create motion object, appending the axis object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));

    /* Request notification of ALL events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);

    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
```

```
    serviceCreate(eventMgr,
                  -1,   /* default (max) priority */
                  -1);  /* -1 => enable interrupts */
meiASSERT(service != NULL);

/* Configure the Position Compare, using a User Limit */
returnValue =
    CompareLimitSet(axis,
                    motor,
                    COMPARE_POSITION,
                    EVENT_TYPE);
msgCHECK(returnValue);

printf("Press any key to exit ...\n");

/* Loop repeatedly */
index       = 0;
motionDone  = TRUE;
while (meiPlatformKey(MPIWaitPOLL) <= 0) {
    MPIEventStatus      eventStatus;
    MEIEventStatusInfo  *info;

    if (motionDone) {
        MPIMotionParams     motionParams;       /* motion parameters */

        /* fill in the MPIMotionParams structure */
        switch (motionType) {
            case MPIMotionTypeS_CURVE: {
                motionParams.sCurve = sCurve[index];
                break;
            }
            case MPIMotionTypeTRAPEZOIDAL: {
                motionParams.trapezoidal = trapezoidal[index];
                break;
            }
            case MPIMotionTypeVELOCITY: {
                motionParams.velocity = velocity[index];
                break;
            }
            default: {
                meiASSERT(FALSE);
                break;
            }
        }

        printf("\n\nMotion Start...");

        /* Start Motion */
        returnValue =
            mpiMotionStart(motion,
                           motionType,
                           &motionParams);
        msgCHECK(returnValue);

        motionDone = FALSE;
```

```
    }

    /* Wait for events */
    returnValue =
        mpiNotifyEventWait(notify,
                           &eventStatus,
                           MPIWaitFOREVER);
    msgCHECK(returnValue);

    info = (MEIEventStatusInfo *)eventStatus.info;

    switch (eventStatus.type) {
        /* In Coarse Event from axis source */
        case MEIEventTypeIN_POSITION_COARSE: {
            printf("\nInCoarse (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeIN_POSITION_FINE: {
            printf("\nInFine (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeAT_TARGET: {
            printf("\nAtTarget (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* Motion Done Event from motion source */
        case MPIEventTypeMOTION_DONE: {
            printf("\nDone (%ld)",
                   info->data.motion.sampleCounter);

            motionDone = TRUE;

            if (++index >= MOTION_COUNT) {
                index = 0;
            }
            break;
        }
        /* User Limit Event from motor source */
        case MEIEventTypeLIMIT_USER0: {
            printf("\nUser Limit #0, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
            break;
        }
        /* User Limit Event from motor source */
        case MEIEventTypeLIMIT_USER1: {
            printf("\nUser Limit #1, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
```

```
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER2: {
        printf("\nUser Limit #2, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER3: {
        printf("\nUser Limit #3, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER4: {
        printf("\nUser Limit #4, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER5: {
        printf("\nUser Limit #5, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER6: {
        printf("\nUser Limit #6, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }
    /* User Limit Event from motor source */
    case MEIEventTypeLIMIT_USER7: {
        printf("\nUser Limit #7, type %d source 0x%x info 0x%x",
                eventStatus.type,
                eventStatus.source,
                eventStatus.info[0]);
        break;
    }

    default: {
        printf("mpiNotifyEventWait() returns 0x%x\n"
            "\teventStatus: type %d source 0x%x info 0x%x\n",
```

```
                    returnValue,
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
        }
    }

    printf("\n");

    /* Delete objects */
    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long CompareLimitSet(MPIAxis axis,
                     MPIMotor motor,
                     long position,
                     MEIEventType eventType)
{
    MEIXmpAxis          *xmpAxis;

    MEIMotorEventConfig motorEventConfig;
    MEIXmpStatus        status;

    long                returnValue;
    MPIEventMask        resetMask;

    returnValue  =
        mpiAxisMemory(axis,
                      (void **)&xmpAxis);

    if (returnValue == MPIMessageOK) {
        returnValue  =
```

```
        mpiAxisMemory(axis,
                      (void **)&xmpAxis);
    }

    mpiEventMaskCLEAR(resetMask);

    if (returnValue == MPIMessageOK) {
        switch (eventType) {
            case MEIEventTypeLIMIT_USER0: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER0);
                break;
            }
            case MEIEventTypeLIMIT_USER1: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER1);
                break;
            }
            case MEIEventTypeLIMIT_USER2: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER2);
                break;
            }
            case MEIEventTypeLIMIT_USER3: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER3);
                break;
            }
            case MEIEventTypeLIMIT_USER4: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER4);
                break;
            }
            case MEIEventTypeLIMIT_USER5: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER5);
                break;
            }
            case MEIEventTypeLIMIT_USER6: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER6);
                break;
            }
            case MEIEventTypeLIMIT_USER7: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER7);
                break;
            }
            default:
                printf("\nError, invalid event type\n");
                break;
        }

        returnValue =
            mpiMotorEventConfigGet(motor,
```

```
                                    (MPIEventType)eventType,
                                    NULL,
                                    &motorEventConfig);
    }

    if (returnValue == MPIMessageOK) {
        /* Set up the Condition[0] block.  The operator between the value in the
            SourceAddress (masked by the Mask), and LimitValue.l, is
            MEIXmpLimitTypeGT (greater than).  Thus, when the value in
            SourceAddress is "GT" (greater than) the value in LimitValue.l,
            then the Condition[0] block is TRUE.
        */
        motorEventConfig.Condition[0].Type = MEIXmpLimitTypeGT;
        motorEventConfig.Condition[0].SourceAddress = &(xmpAxis->ActualPosition);
        motorEventConfig.Condition[0].Mask = 0xffffffff;
        motorEventConfig.Condition[0].LimitValue.l = position;

        /* Set up the Condition[1] block.  The operator is MEIXmpLimitTypeFALSE.
            Therefore, the Condition[1] block is always FALSE.
        */
        motorEventConfig.Condition[1].Type = MEIXmpLimitTypeFALSE;
        motorEventConfig.Condition[1].SourceAddress = NULL;
        motorEventConfig.Condition[1].Mask = 0;
        motorEventConfig.Condition[1].LimitValue.l = 0;

        motorEventConfig.Status = status;
        /* Determine logic result from Condition[0] only */
        motorEventConfig.Logic  = MEIXmpLogicSINGLE;

        /* Disable the output feature */
        motorEventConfig.Output.OutputPtr = NULL;
        motorEventConfig.Output.AndMask = 0x0;
        motorEventConfig.Output.OrMask = 0x0;
        motorEventConfig.Output.Enabled = 0;

        returnValue =
            mpiMotorEventConfigSet(motor,
                                    (MPIEventType)eventType,
                                    NULL,
                                    &motorEventConfig);
    }

    /* Reset event in case conditions have already been satisfied */
    if (returnValue == MPIMessageOK) {
        returnValue =
                mpiMotorEventReset(motor,
                                    resetMask);
    }

    return returnValue;
}
```

usrlim3.c -- Position comparison using User Limits to generate events and outputs.

```
/* UsrLim3.c */

/* Copyright(c) 1991-2002 by Motion Engineering, Inc.  All rights reserved.
 *
 * This software  contains proprietary and  confidential information  of
 * Motion Engineering Inc., and its suppliers.  Except as may be set forth
 * in the license agreement under which  this software is supplied, use,
 * disclosure, or  reproduction is prohibited without the prior express
 * written consent of Motion Engineering, Inc.
 */

/*

:Position comparison using User Limits to generate events and outputs.

This sample code shows how to configure the XMP controller's User Limits
 to compare an axis' actual position to a specific position value.  If the
 actual position is greater than or equal to the value specified, then
 the controller will generate a User Event to the Host.  Additionally,
 a Transceiver is configured for an output.  When the User Limit triggers
 an event, an output bit will be set or cleared.

The function CompareLimitSet(...) demonstrates how to configure a
 User Limit for position compare.  There are two positions compared,
 "positionEnable" and "positionDisable" which specify the range in which
 the output is written.  The output bit is specified in the "outputMask"
 and the state is specified by "enableOutput".

Note, each User Limit configuration can only apply a single output AND and
 OR mask.  Thus, to set an output requires a User Limit, and to clear
 an output requires another User Limit.  Be careful not to configure the
 position compare values that would cause more than one User Limit to write
 to the output simultaneously.


The XMP-Series controller "User Limit" feature allows the user to program a
 the result of two logical condition to generate an event to the host.  Also,
 a "User Limit" can be configured to write an "output" to any XMP memory
 location, using an AND mask and OR mask.

Here is the "User Limit" block structure:

typedef struct {
    MEIXmpLimitType         Type;
    void                    *SourceAddress;
    long                    Mask;
    MEIXmpGenericValue      LimitValue;
} MEIXmpLimitCondition;

typedef struct {
    long                    AndMask;
    long                    OrMask;
    long                    *OutputPtr;
    long                    Enabled;
```

```
} MEIXmpLimitOutput;

typedef struct {
    MEIXmpLimitCondition    Condition[MEIXmpLimitConditions];
    MEIXmpStatus            Status;
    MEIXmpLogic             Logic;
    MEIXmpLimitOutput       Output;
    long                    Count;
    long                    State;
} MEIXmpLimitData;


MEIXmpLimitTypes are the operators that are used for the User Limit's
 Condition[0] and Condition[1].  They are found in xmp.h.

*SourceAddress is a pointer to an XMP memory location.

Mask is ANDed with the value located at the *SourceAddress.

LimitValue is compared with the masked value located at *SourceAddress,
 using the Type operator.

Status defines the status bit used to generate an event to the host.

MEIXmpLogic is the logic applied between the two condition block outputs,
 Condition[0] and Condition[1]:

 MEIXmpLogicNEVER
    Does NOT evaluate Condition[0], Condition[1].  No event is generated.

 MEIXmpLogicSINGLE
    Only evaluates Condition[0].  Event is generated if Condition[0] is TRUE.

 MEIXmpLogicOR
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    OR Condition[1]) = TRUE.

 MEIXmpLogicAND
    Evaluates Condition[0], Condition[1].  Event is generated if (Condition[0]
    AND Condition[1]) = TRUE.

The other MEIXmpLogic enums in xmp.h are for internal use only.

*OutputPtr is a pointer to an XMP memory location.

AndMask is ANDed, and OrMask is ORed with the value located at
 *OutputPtr when the resultant MEIXmpLogic applied Condition[0] and
 Condition[1] are TRUE.

Count and State are for internal use only.  Do not write these values.
  The MPI method, mpiMotorEventConfigSet(...) will not write these values.


The XMP supports up to 8 User Limits per motor.  The User Limit processing
 occurs in the firmware background task.  For maximum efficiency, the XMP only
 processes the User Limits (in order 0, 1, 2, etc.), up to the largest User
 Limit number that has (motorEventConfig.Logic ! = MEIXmpLogicNEVER).
```

```
For example:
 If UserLimit 0 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    only process the first UserLimit for each motor.
 If UserLimit 3 (motor 0) is configured for MEIXmpLogicSINGLE, then the XMP will
    process UserLimits number 0, 1, 2, and 3 for each motor.

It's best to use the lower numbered UserLimits for maximum efficiency.  When
 finished using UserLimits, it's a good idea to set the Logic to
 MEIXmpLogicNEVER.

Warning!  This is a sample program to assist in the integration of the
 XMP motion controller with your application.  It may not contain all
 of the logic and safety features that your application requires.
*/

#include <stdlib.h>
#include <stdio.h>

#include "stdmpi.h"
#include "stdmei.h"

#include "apputil.h"

#if defined(ARG_MAIN_RENAME)
#define main    UsrLim3Main

argMainRENAME(main, UsrLim3)
#endif

#define MOTION_COUNT        (2)

/* Motor I/O Configurations */
#define OUTPUT_BIT      (MEIMotorTransceiverIdA)    /* A, B, or C */
#define OUTPUT_MASK     (MEIMotorTransceiverMaskA)  /* mask for output bit */
#define OUTPUT_INVERT   (TRUE)                      /* inverted or normal */


/* Compare positions to enable/disable output */
long comparePosition[] = {
    1000,   /* enable */
    2000,   /* disable */
    5000,
    6000,
    10000,
    11000,
    15000,
    16000   /* disable */
};

/* User Limits for compare positions */
MEIEventType eventType[] = {
    MEIEventTypeLIMIT_USER0,
    MEIEventTypeLIMIT_USER1,
    MEIEventTypeLIMIT_USER2,
    MEIEventTypeLIMIT_USER3,
    MEIEventTypeLIMIT_USER4,
```

```
    MEIEventTypeLIMIT_USER5,
    MEIEventTypeLIMIT_USER6
};

#define LIMIT_COUNT (sizeof(eventType) / sizeof(long))


/* Command line arguments and defaults */
long            axisNumber      = 0;
long            motionNumber    = 0;
long            motorNumber     = 0;
MPIMotionType   motionType      = MPIMotionTypeS_CURVE;

Arg argList[] = {
    {   "-axis",    ArgTypeLONG,    &axisNumber,    },
    {   "-motion",  ArgTypeLONG,    &motionNumber,  },
    {   "-motor",   ArgTypeLONG,    &motorNumber,   },
    {   "-type",    ArgTypeLONG,    &motionType,    },

    {   NULL,       ArgTypeINVALID, NULL,   }
};

double position[MOTION_COUNT] = {
    25000.0,
    0.0,
};

MPITrajectory trajectory[MOTION_COUNT] = {
    /* velocity     accel       decel       jerkPercent */
    { 1000.0,       1000000.0,  1000000.0,  0.0,    },
    { 10000.0,      100000.0,   100000.0,   0.0,    },
};

/* Motion Parameters */
MPIMotionSCurve sCurve[MOTION_COUNT] = {
    {   &trajectory[0], &position[0],   },
    {   &trajectory[1], &position[1],   },
};

MPIMotionTrapezoidal    trapezoidal[MOTION_COUNT] = {
    {   &trajectory[0], &position[0],   },
    {   &trajectory[1], &position[1],   },
};

MPIMotionVelocity   velocity[MOTION_COUNT] = {
    {   &trajectory[0], },
    {   &trajectory[1], },
};


/* Function Prototypes */
long CompareLimitSet(MPIAxis axis,
                     MPIMotor motor,
                     long positionEnable,
                     long positionDisable,
                     long outputMask,
                     long enableOutput,
```

```
                    MEIEventType eventType);

int
    main(int    argc,
         char   *argv[])
{
    MPIControl  control;    /* motion controller handle */
    MPIAxis     axis;       /* axis object */
    MPIMotion   motion;     /* motion object */
    MPIMotor    motor;      /* motor object */
    MPINotify   notify;     /* event notification object */
    MPIEventMgr eventMgr;   /* event manager handle */

    MPIEventMask    eventMask;
    MEIMotorConfig  motorConfigXmp;  /* XMP motor I/O configuration */

    long    returnValue;    /* return value from library */

    long    index;
    long    motionDone;     /* flag when Done occurs */

    Service service;

    MPIControlType      controlType;
    MPIControlAddress   controlAddress;

    long    argIndex;

    /* Parse command line for Control type and address */
    argIndex =
        argControl(argc,
                   argv,
                   &controlType,
                   &controlAddress);

    /* Parse command line for application-specific arguments */
    while (argIndex < argc) {
        long    argIndexNew;

        argIndexNew = argSet(argList, argIndex, argc, argv);

        if (argIndexNew <= argIndex) {
            argIndex = argIndexNew;
            break;
        }
        else {
            argIndex = argIndexNew;
        }
    }

    /* Check for unknown/invalid command line arguments */
    if ((argIndex < argc) ||
        (axisNumber >= MEIXmpMAX_Axes) ||
        (motionNumber >= MEIXmpMAX_MSs) ||
        (motorNumber >= MEIXmpMAX_Motors) ||
        (motionType < MPIMotionTypeFIRST) ||
        (motionType >= MEIMotionTypeLAST)) {
```

```
        meiPlatformConsole("usage: %s %s\n"
                           "\t\t[-axis # (0 .. %d)]\n"
                           "\t\t[-motion # (0 .. %d)]\n"
                           "\t\t[-motor # (0 .. %d)]\n"
                           "\t\t[-type # (0 .. %d)]\n",
                            argv[0],
                            ArgUSAGE,
                            MEIXmpMAX_Axes - 1,
                            MEIXmpMAX_MSs - 1,
                            MEIXmpMAX_Motors - 1,
                            MEIMotionTypeLAST - 1);
        exit(MPIMessageARG_INVALID);
    }

    switch (motionType) {
        case MPIMotionTypeS_CURVE:
        case MPIMotionTypeTRAPEZOIDAL:
        case MPIMotionTypeVELOCITY: {
            break;
        }
        default: {
            meiPlatformConsole("%s: %d: motion type not available\n",
                               argv[0],
                               motionType);
            exit(MPIMessageUNSUPPORTED);
            break;
        }
    }

    /* Create motion controller object */
    control =
        mpiControlCreate(controlType,
                         &controlAddress);
    msgCHECK(mpiControlValidate(control));

    /* Initialize motion controller */
    returnValue = mpiControlInit(control);
    msgCHECK(returnValue);

    /* Create axis object for axisNumber */
    axis =
        mpiAxisCreate(control,
                      axisNumber);
    msgCHECK(mpiAxisValidate(axis));

    /* Create motor object for motorNumber */
    motor =
        mpiMotorCreate(control,
                       motorNumber);
    msgCHECK(mpiMotorValidate(motor));

    /* Create motion object, appending the axis object */
    motion =
        mpiMotionCreate(control,
                        motionNumber,
                        axis);
    msgCHECK(mpiMotionValidate(motion));
```

```
    /* Request notification of ALL events from motion */
    mpiEventMaskCLEAR(eventMask);
    mpiEventMaskALL(eventMask);
    meiEventMaskALL(eventMask);

    returnValue =
        mpiMotionEventNotifySet(motion,
                                eventMask,
                                NULL);
    msgCHECK(returnValue);

    /* Create event notification object for motion */
    notify =
        mpiNotifyCreate(eventMask,
                        motion);
    msgCHECK(mpiNotifyValidate(notify));

    /* Create event manager object */
    eventMgr = mpiEventMgrCreate(control);
    msgCHECK(mpiEventMgrValidate(eventMgr));

    /* Add notify to event manager's list */
    returnValue =
        mpiEventMgrNotifyAppend(eventMgr,
                                notify);
    msgCHECK(returnValue);

    /* Create service thread */
    service =
        serviceCreate(eventMgr,
                      -1,   /* default (max) priority */
                      -1);  /* -1 => enable interrupts */
    meiASSERT(service != NULL);

    /* Configure the specified OUTPUT_BIT */
    returnValue =
        mpiMotorConfigGet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    motorConfigXmp.Transceiver[OUTPUT_BIT].Config = MEIMotorTransceiverConfigOUTPUT;
    motorConfigXmp.Transceiver[OUTPUT_BIT].Invert = OUTPUT_INVERT;

    returnValue =
        mpiMotorConfigSet(motor,
                          NULL,
                          &motorConfigXmp);
    msgCHECK(returnValue);

    /* Configure the Position Compares, using User Limits */
    for (index = 0; index < (long)LIMIT_COUNT; index++) {
        long enableOutput = ((index % 2) ? FALSE:TRUE);

        printf("\nEn:%d", enableOutput);
```

```
        returnValue =
            CompareLimitSet(axis,
                            motor,
                            comparePosition[index],
                            comparePosition[index + 1],
                            OUTPUT_MASK,
                            enableOutput,
                            eventType[index]);
        msgCHECK(returnValue);
    }

    printf("Press any key to exit ...\n");

    /* Loop repeatedly */
    index       = 0;
    motionDone  = TRUE;
    while (meiPlatformKey(MPIWaitPOLL) <= 0) {
        MPIEventStatus      eventStatus;
        MEIEventStatusInfo  *info;

        if (motionDone) {
            MPIMotionParams     motionParams;       /* motion parameters */

            /* fill in the MPIMotionParams structure */
            switch (motionType) {
                case MPIMotionTypeS_CURVE: {
                    motionParams.sCurve = sCurve[index];
                    break;
                }
                case MPIMotionTypeTRAPEZOIDAL: {
                    motionParams.trapezoidal = trapezoidal[index];
                    break;
                }
                case MPIMotionTypeVELOCITY: {
                    motionParams.velocity = velocity[index];
                    break;
                }
                default: {
                    meiASSERT(FALSE);
                    break;
                }
            }

            printf("\n\nMotion Start...");

            /* Start Motion */
            returnValue =
                mpiMotionStart(motion,
                               motionType,
                               &motionParams);
            msgCHECK(returnValue);

            motionDone = FALSE;
        }

        /* Wait for events */
        returnValue =
```

```
        mpiNotifyEventWait(notify,
                           &eventStatus,
                           MPIWaitFOREVER);
    msgCHECK(returnValue);

    info = (MEIEventStatusInfo *)eventStatus.info;

    switch (eventStatus.type) {
        /* In Coarse Event from axis source */
        case MEIEventTypeIN_POSITION_COARSE: {
            printf("\nInCoarse (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeIN_POSITION_FINE: {
            printf("\nInFine (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* In Fine Event from axis source */
        case MEIEventTypeAT_TARGET: {
            printf("\nAtTarget (%ld)",
                   info->data.axis.sampleCounter);
            break;
        }
        /* Motion Done Event from motion source */
        case MPIEventTypeMOTION_DONE: {
            printf("\nDone (%ld)",
                   info->data.motion.sampleCounter);

            motionDone = TRUE;

            if (++index >= MOTION_COUNT) {
                index = 0;
            }
            break;
        }
        /* User Limit Event from motor source */
        case MEIEventTypeLIMIT_USER0: {
            printf("\nUser Limit #0, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
            break;
        }
        /* User Limit Event from motor source */
        case MEIEventTypeLIMIT_USER1: {
            printf("\nUser Limit #1, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
            break;
        }
        /* User Limit Event from motor source */
        case MEIEventTypeLIMIT_USER2: {
            printf("\nUser Limit #2, type %d source 0x%x info 0x%x",
```

```
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
            /* User Limit Event from motor source */
            case MEIEventTypeLIMIT_USER3: {
                printf("\nUser Limit #3, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
            /* User Limit Event from motor source */
            case MEIEventTypeLIMIT_USER4: {
                printf("\nUser Limit #4, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
            /* User Limit Event from motor source */
            case MEIEventTypeLIMIT_USER5: {
                printf("\nUser Limit #5, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
            /* User Limit Event from motor source */
            case MEIEventTypeLIMIT_USER6: {
                printf("\nUser Limit #6, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
            /* User Limit Event from motor source */
            case MEIEventTypeLIMIT_USER7: {
                printf("\nUser Limit #7, type %d source 0x%x info 0x%x",
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }

            default: {
                printf("mpiNotifyEventWait() returns 0x%x\n"
                    "\teventStatus: type %d source 0x%x info 0x%x\n",
                    returnValue,
                    eventStatus.type,
                    eventStatus.source,
                    eventStatus.info[0]);
                break;
            }
        }
    }
```

```
    printf("\n");

    /* Delete objects */
    returnValue = mpiMotorDelete(motor);
    msgCHECK(returnValue);

    returnValue = mpiMotionDelete(motion);
    msgCHECK(returnValue);

    returnValue = mpiAxisDelete(axis);
    msgCHECK(returnValue);

    returnValue = serviceDelete(service);
    msgCHECK(returnValue);

    returnValue = mpiEventMgrDelete(eventMgr);
    msgCHECK(returnValue);

    returnValue = mpiNotifyDelete(notify);
    msgCHECK(returnValue);

    returnValue = mpiControlDelete(control);
    msgCHECK(returnValue);

    return ((int)returnValue);
}

long CompareLimitSet(MPIAxis axis,
                     MPIMotor motor,
                     long positionEnable,
                     long positionDisable,
                     long outputMask,
                     long enableOutput,
                     MEIEventType eventType)
{
    MPIControl          control;
    MEIXmpAxis          *xmpAxis;
    MEIXmpData          *firmware;

    MEIMotorEventConfig motorEventConfig;
    MEIXmpStatus        status;

    long                motorIndex;
    long                returnValue;
    MPIEventMask        resetMask;

    /* Determine motor number */
    returnValue =
        mpiMotorNumber(motor,
                       &motorIndex);

    /* Determine control handle */
    if(returnValue == MPIMessageOK) {
        control = mpiMotorControl(motor);
        returnValue = mpiControlValidate(control);
    }
```

```
    /* Get pointer to XMP firmware */
    if(returnValue == MPIMessageOK) {
        returnValue =
            mpiControlMemory(control,
                             &(void *)firmware,
                             NULL);
    }

    if (returnValue == MPIMessageOK) {
        returnValue  =
            mpiAxisMemory(axis,
                          &(void *)xmpAxis);
    }

    if (returnValue == MPIMessageOK) {
        returnValue  =
            mpiAxisMemory(axis,
                          &(void *)xmpAxis);
    }

    mpiEventMaskCLEAR(resetMask);

    if (returnValue == MPIMessageOK) {
        switch (eventType) {
            case MEIEventTypeLIMIT_USER0: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER0);
                break;
            }
            case MEIEventTypeLIMIT_USER1: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER1);
                break;
            }
            case MEIEventTypeLIMIT_USER2: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER2);
                break;
            }
            case MEIEventTypeLIMIT_USER3: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER3);
                break;
            }
            case MEIEventTypeLIMIT_USER4: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER4);
                break;
            }
            case MEIEventTypeLIMIT_USER5: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER5);
                break;
            }
            case MEIEventTypeLIMIT_USER6: {
                status = MEIXmpStatusLIMIT;
```

```
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER6);
                break;
            }
            case MEIEventTypeLIMIT_USER7: {
                status = MEIXmpStatusLIMIT;
                mpiEventMaskSET(resetMask, MEIEventTypeLIMIT_USER7);
                break;
            }
            default:
                printf("\nError, invalid event type\n");
                break;
        }

        returnValue =
            mpiMotorEventConfigGet(motor,
                                   (MPIEventType)eventType,
                                   NULL,
                                   &motorEventConfig);
    }

    if (returnValue == MPIMessageOK) {
        /* Write to Output between positionEnable and positionDisable */
        if (positionDisable > positionEnable ) {
            motorEventConfig.Condition[0].Type = MEIXmpLimitTypeGE;
            motorEventConfig.Condition[1].Type = MEIXmpLimitTypeLT;
        }

        /* Write to Output between positionDisable and positionEnable */
        if (positionDisable <= positionEnable ) {
            motorEventConfig.Condition[0].Type = MEIXmpLimitTypeLT;
            motorEventConfig.Condition[1].Type = MEIXmpLimitTypeGE;
        }

        if (enableOutput) {
            /* Enable output */
            motorEventConfig.Output.AndMask = 0xFFFFFFFF;
            motorEventConfig.Output.OrMask = outputMask;
        }
        else {
            /* Disable Output */
            motorEventConfig.Output.AndMask = (~outputMask);
            motorEventConfig.Output.OrMask = 0x0;
        }

        /* Set up the Condition[0] block. */
        motorEventConfig.Condition[0].SourceAddress = &(xmpAxis->ActualPosition);
        motorEventConfig.Condition[0].Mask = 0xffffffff;
        motorEventConfig.Condition[0].LimitValue.l = positionEnable;

        /* Set up the Condition[1] block. */
        motorEventConfig.Condition[1].SourceAddress = &(xmpAxis->ActualPosition);
        motorEventConfig.Condition[1].Mask = 0xffffffff;
        motorEventConfig.Condition[1].LimitValue.l = positionDisable;

        motorEventConfig.Status = status;
        /* Determine logic result from Condition[0] AND Condition[1] */
        motorEventConfig.Logic  = MEIXmpLogicAND;
```

```
        motorEventConfig.Output.OutputPtr =
            &firmware->Motor[motorIndex].IO.MotorOutput;    /* output address */
        motorEventConfig.Output.Enabled = TRUE;

        returnValue =
            mpiMotorEventConfigSet(motor,
                                   (MPIEventType)eventType,
                                   NULL,
                                   &motorEventConfig);
    }

    /* Reset event in case conditions have already been satisfied */
    if (returnValue == MPIMessageOK) {
        returnValue =
                mpiMotorEventReset(motor,
                                   resetMask);
    }

    return returnValue;
}
```