```cpp
// ----------------------------------------------------------------------------
//
// This file contains the main code - OmXyzDll.cpp - to build a DLL that controls a motorized stage with up to 6 degrees of freedom
//
// The motors must be controlled by a NanoArduino with a specific scheme that is described
// in the user manual of this DLL: "Universal DLL.pdf".
//
//
// Manuel Fortunato (MF), May 2021
// ----------------------------------------------------------------------------



#pragma hdrstop
#include <time.h>
#include<Windows.h>
#include <winuser.h>
#include <vector>
#include<math.h>
#include <cmath>
#include <stdio.h>
#include <stdlib>
#define XYZDLL_EXPORTS 1
#include "OmXyzDll.h"
#include "rs232.h"
#include <cstring>
#include <string>
#include <sstream>

// ----------------------------------------------------------------------------
#pragma package(smart_init)

// ______Global variables____________________________
//

double CurrentDllPosition[3];
double CurrentDllAngle[3];
double DemandPosition[3];
double DemandAngle[3];
double PosStep[3];
double AngleStep[3];
double LinSpeed[3];
double RotSpeed[3];
clock_t tLin;
clock_t tRot;
bool DllPowerOn;
#define nOptions 9
char OptionText[nOptions][32]={0};
bool optionsCopied = false;



/*MF: Global variables are useful since they can be accessed and modified by any function.
 Besides the global variables included originally in the code provided with the installation of OMDAQ-3,
 the following global variables were added. */
int PosDOF, AngleDOF;    //MF: Global variables to store the number of degrees of freedom for linear and rotational motion
bool Axis[6];   //MF: Global variable to check which axes are available for this particular stage

//MF: Global variables to store RS232 communication parameters
char modo[4];
int taxabaud;
int port_nmr;

```

```
//MF: Global variables for debbuging and testing purposes
bool port=true; /*MF: variable used to prevent RS-232 communications just to test the DLL without the hardware
if false no RS232 orders are sent and there's no error when linking OMDAQ-3 with the DLL
without the actual RS232 connection established */
bool show_orders=false; /*MF: variable used to show the motion orders in a pop up window. When set to true
the strings sent to the NanoArduino appear in a pop up window.*/

using namespace std;



//MF: added function
//MF: function to check if char array has only numbers
//MF: used to check if parameters given in the OMDAQ-3 parameters window are okay
bool ISnumber(char *c) {
    string c_s(c);
    bool has_only_digits = (c_s.find_first_not_of( "0123456789." ) == string::npos);
    return has_only_digits;
};





/******************************* Adminstration routines *******************************/




/*XyzCapabilityMask returns a DWORD mask that describes the basic functionality
 of the hardware and allows OMDAQ to make the user interface.
 The return value is assembled from the capability constants
 defined in OmXyzDll_StatusBits.h */
 /*>>>>>> THIS MUST BE DEFINED <<<<<<*/
XYZ_DLL DWORD _CALLSTYLE_ XyzCapabilityMask() {
    unsigned long CAP=0;
    //MF: motion along the 3 cartesian axis must always be declared (even if the stage is not able to perform such motion)
    CAP= CAP | XYZCAP_XYZ3;

    /*MF: When OMDAQ-3 is executed there is a window that pops up asking for parameters. The user inputs the parameters to
     set up the RS232 communication channel and provides the step for each available axis of the stage.
      After the parameters have been provided and the pop up window is closed the initialisation function
      XyzInitialise(char **options, int szOptions) is called and the parameters are retrieved through the char **options argument.
      This function - XyzCapabilityMask() - is called, for some reason, before and after the initialisation function. In the case
      of this stage in particular the available axes are only known after the options have been retrieved, i.e., copied, which is
      indicated by the bool optionsCopied.*/

    //MF: if optionsCopied==true then the initialize function was already called and we already know the number of degrees of freedom
    if(optionsCopied) {

        if(AngleDOF==1) {
            CAP= CAP | XYZCAP_ROT1;
        }
        else if(AngleDOF==2) {
            CAP= CAP | XYZCAP_ROT2;
        }
        else if(AngleDOF==3) {
            CAP= CAP | XYZCAP_ROT3;
        }
    }

    return CAP;
}


// XyzDllVersion returns the version numbers of the DLL file.
XYZ_DLL bool _CALLSTYLE_ XyzDllVersion(int * majorVersion, int * minorVersion,
    int * buildNumber) {
  *majorVersion = 1;
  *minorVersion = 0;
  *buildNumber = 12;
  return true;
}

/* XyzDescription fills a char string that describes the XYZ stage
 nChar is the length of the supplied buffer (typically 80 characters)*/
XYZ_DLL bool _CALLSTYLE_ XyzDescription(char *statusText, int nChar) {
  strncpy(statusText, "XYZ stage controlled by user-supplied DLL", nChar);
  return true;
}

/* XyzHwDescription fills a char string that decsribes the current setup
 (COM ports, card slot numbers etc.)
 nChar is the length of the supplied buffer. (typically 80 characters) */
XYZ_DLL bool _CALLSTYLE_ XyzHwDescription(char *statusText, int nChar) {
  strncpy(statusText, "COM45 9600baud", nChar);
  return true;
}


/* XyzAuthor returns the author credits and copyrights etc.
 nChar is the length of the supplied buffer.  (typically 80 characters) */
XYZ_DLL bool _CALLSTYLE_ XyzAuthor(char *statusText, int nChar) {
  strncpy(statusText,
      "DLL written by Manuel Fortunato, 2021", nChar);
  return true;
}




// ---------Procedures for optional parameters ----------
/* When OMDAQ-3 is executed there is a window that pops up asking for parameters.
 These parameters are passed as strings to the XyzInitialise(...) procedure through
 the options argument.

 In order to create the window interface OMDAQ needs to know the number of
 parameters and the name of each one. These are obtained using the XyzOptionCount
 and XyzOptionHeader procedures.

 XyzOptionValue establishes the default parameter values. This is used to
 provide sensible starting values for the parameters to assist the user in
 setting up a new stage.
 */
XYZ_DLL int _CALLSTYLE_ XyzOptionCount() {
  return nOptions;
}

//These allow the DLL to get the parameter filename and the DDL folder from OMDAQ
XYZ_DLL bool _CALLSTYLE_ XyzSetParameterFileName(wchar_t *cText, int nChar) {
  // IniFile = UnicodeString(&cText[0]);
  return true;
}

XYZ_DLL bool _CALLSTYLE_ XyzSetDLLfolder(wchar_t *statusText, int nChar) {
  return true;
```

```
}





/* XyzOptionHeader sets up the name of the parameters in the parameters window interface. */
/*MF: added options to set up RS232 communication and to get the degrees of freedom of the stage. */
XYZ_DLL bool _CALLSTYLE_ XyzOptionHeader(int nHdr, char * optionsHdr,
    int szOptionsHdr) {



  bool ok = false;
  char * initHdrs[nOptions] = {"COM", "Baud", "Mode", "Step (X)", "Step (Y)", "Step (Z)", "Step (rot1)", "Step (rot2)", "Step (rot3)"}; // For example...
  if ((nHdr >= 0) && (nHdr < nOptions)) {
    strncpy(optionsHdr, initHdrs[nHdr], szOptionsHdr);
    ok = true;
  }
  return ok;
}




/* XyzOptionValue sets the default parameter values in the parameters window interface
to assist the user in setting up a new stage.
 Should return false if nHdr is out of range. */
/* MF: added parameters for a 2 axis translation stage with a 0.00254 mm step in each axis
  according to a stage that exists in Campus Tecnológico e Nuclear*/
XYZ_DLL bool _CALLSTYLE_ XyzOptionValue(int nHdr, char * optionVal,
    int szOptionVal) {


  bool ok = false;


  char * initVals[nOptions] = {"5", "9600", "8N1", "0.00254" , "0.00254", "missing", "missing", "missing", "missing"}; // For example...
  if ((nHdr >= 0) && (nHdr < nOptions)) {
    if (!optionsCopied) {
      strncpy(&OptionText[nHdr][0], initVals[nHdr], 32*sizeof(char));
    }
    strncpy(optionVal, &OptionText[nHdr][0], szOptionVal);
    ok = true;
  }


  return ok;
}






/****************************** End of administration routines ******************************/




/****************************** Initialisation routines ******************************/

```

```

// ----------------------------------------------------------------------------
XYZ_DLL bool _CALLSTYLE_ XyzInitialise(char **options, int szOptions) {
  /* Initialisation code here

   This function obtains the parameter values from the parameters window interface
   through the char **options argument. Namely the RS232 communication parameters
   and the step of each avaliable axis. Global variables are defined with these parameters.

   The RS232 channel is opened.

    */



    /*MF: added a pop up warning that is launched when OMDAQ-3 is executed
    so that the user is careful when setting up the parameters of the stage.
    Wrong steps can result in stage damage if hardware limits are breached */
    int msgboxID = MessageBoxA(
            NULL,
            "Please make sure that the step provided for each axis in the parameters window coincides with the step of the stepping motor that performs the motion. Incorrect values will result in a mismatch between the position of the stage and the position perceived by OMDAQ and possibly damage to the stage if hardware limits are inadvertently breached.\nFor each axis, the units used for \"step\" will be the units used to display the position in OMDAQ. OMDAQ informs the user that the linear positions are in millimeters and that the rotational positions are in degrees. That's NOT NECESSARILY the case. However the use of these units is ADVISED to reduce the probability of damaging the equipment due to a user's mistake.\n If the stage does not have a degree of freedom, just write \"missing\" in the corresponding step.",
            "WARNING - motorized stage",
            MB_TOPMOST | MB_ICONWARNING | MB_OK
        );


    if (szOptions != nOptions) {
        return false;
    }


    for (int i = 0; i < szOptions; ++i) {
        ZeroMemory(&OptionText[i][0], 32*nOptions*sizeof(char));
        strcpy(&OptionText[i][0], options[i]);
    }
    optionsCopied = true;





    for (int i = 0; i < 3; ++i) {
        CurrentDllPosition[i] = 0;
        CurrentDllAngle[i] = 0;
    }






    //MF: checking the used axis according to the parameters given in the parameters window interface
    //MF: setting variables bool Axis[6] with the used axes and int PosDOF with the number of linear degrees of freedom
    double LinSteps[3];
    double RotSteps[3];
```

```
      PosDOF=0;
      for (int i=0; i < 3; i++) {
          if(ISnumber(options[3+i])){
              PosStep[i]=atof(options[3+i]);
              PosDOF+=1;
              Axis[i]=true;
          }
          else {
              PosStep[i]=0;
              Axis[i]=false;
          }
      }


       //MF: setting variables bool Axis[6] with the used axes and int AngleDOF with the number of rotational degrees of freedom
      AngleDOF=0;
      int k=0;
      for (int i = 0; i < 3; i++) {
            if(ISnumber(options[6+i])) {
              AngleStep[k]=atof(options[6+i]);
              AngleDOF+=1;
              Axis[k+3]=true;
              k=k+1;
          }
      }

      for (int i=k; i<3; i++){
          AngleStep[i]=0;
          Axis[i+3]=false;
      }




    //MF: opening COM port
    //MF: getting RS232 parameters from OMDAQ-3 parameters window

      if(options[0]!=NULL){
      port_nmr=atoi(options[0])-1;
      }

      if(options[1]!=NULL){
      taxabaud=atoi(options[1]);
      }

      if(options[2]!=NULL){
      std::strcpy(modo,options[2]);
      }


    /*MF: openning communications port. If something goes wrong
    a pop up message appears saying that there was an error */
      if(port){


          if(RS232_OpenComport(port_nmr, taxabaud, modo))   {

              int msgboxID3 = MessageBoxA(
                  NULL,
                  "It was not possible to open the communication channel between OMDAQ3 and the NanoArduino. Please check that the parameters \"COM\", \"Baud\" and \"Mode\" are set correctly in the parameters window. The default value for \"Baud\" is 9600 and for \"Mode\" is 8N1",
                  "ERROR",
                  MB_TOPMOST | MB_ICONERROR | MB_OK
              );

```

```

        return(0);

        }


    }



    DllPowerOn = true;
    return true;

}

/*---------------------------------------------------------------------------*/
XYZ_DLL bool _CALLSTYLE_ XyzShutDown() {
  /* Full shutdown code here  - stop stage if it's moving,
   power down, free comms links and free resources.

   OMDAQ saves the position at shutdown ready for the next startup.
   return false if it fails. */
  return true;
}

/*--------------------------------------------------------------------------*/
/* This procedure initialises the values of the position readouts to the supplied values.
 NewPosition is a pointer to a double[3] array which contains the
 new values of the absolute postions for axes 0..2 */
XYZ_DLL bool _CALLSTYLE_ XyzSetCurrentPosition(double * NewPosition) {

  /*MF: if a linear axis is available the linear position variables are updated with whatever value
  intended. If not the linear position variables are set to 0 */
  for (int i = 0; i < 3; ++i) {
    if(Axis[i]){
        CurrentDllPosition[i] = NewPosition[i];
        DemandPosition[i] = NewPosition[i];   }
    else if(!Axis[i]){
        CurrentDllPosition[i]=0;
        DemandPosition[i]=0;  }

  }

  return true;
}


/* This procedure initialises the values of the angle readouts to the supplied values.
NewAngle is a pointer to a double[3] array which contains the
 new values of the absolute angles for axes 3..5 */
XYZ_DLL bool _CALLSTYLE_ XyzSetCurrentAngle(double * NewAngle) {

  /*MF: if a rotational axis is available the angular position variables are updated with whatever value
  intended. If not the angular position variables are set to 0 */
  for (int i = 0; i < 3; ++i) {
    if(Axis[i+3]){
        CurrentDllAngle[i] = NewAngle[i];
        DemandAngle[i] = NewAngle[i];   }
    else if(!Axis[i+3]){
        CurrentDllAngle[i]=0;
        DemandAngle[i]=0;  }


  }
  return true;
}
```

```

/********************************** End of initialisation routines *******************************/






/********************************** Routines to set up motion parameters *******************************/

/* XyzSetSpeed(...) and XyzSetAccel(...) set the LINEAR speed and acceleration per axis,
respectively. The acceleration is assumed to be the same in the accel and decel phases.  Units are  mm/sec and mm/sec2.
NewAccel and NewSpeed are pointers to double[3] arrays containing the new values for each axis.
At present OMDAQ only allows a single accel value for all axes. */
XYZ_DLL bool _CALLSTYLE_ XyzSetAccel(double * NewAccel) {
    /*This was not a needed funcionality
    so this function was not used. */
  return true;
}

XYZ_DLL bool _CALLSTYLE_ XyzSetSpeed(double * NewSpeed) {

    /*This was not a needed funcionality
    so this function was not used.*/

  for (int i = 0; i < 3; ++i) {
    LinSpeed[i] = NewSpeed[i];
  }

  return true;
}

/* XyzSetRotSpeed(...) and XyzSetRotAccel set the ROTATIONAL speed and acceleration per axis, respectively.
 The acceleration is assumed to be the same in the accel and decel phases.  Units are  deg/sec and deg/sec2
 NewAccel and NewSpeed are pointers to double[3] arrays containing the new values for each axis.   */
XYZ_DLL bool _CALLSTYLE_ XyzSetRotAccel(double * NewAccel) {

    /*This was not a needed funcionality
    so this function was not used.*/

  return true;
}

XYZ_DLL bool _CALLSTYLE_ XyzSetRotSpeed(double * NewSpeed) {

    /*This was not a needed funcionality
    so this function was not used.*/

  for (int i = 0; i < 3; ++i) {
    RotSpeed[i] = NewSpeed[i];
  }
  return true;
}

/* XyzPowerOn(...) is meant to control the power of the stage (on or off).
 If Enabled = true (false) the power should be turned ON (OFF) for all axes.
 It should leave the controller active and reporting.
 Returns true for success. */
XYZ_DLL bool _CALLSTYLE_ XyzPowerOn(bool Enabled) {

    /*This was not a needed funcionality
    so this function was not used.*/

  DllPowerOn = Enabled;
  return true;
}



/*********************************** End of routines to set up motion parameters *******************************/







/*********************************** Routines for motion command ********************************************/


/*MF: To understand the format of the strings sent to the Arduino Nano please read section 2 . "Communication with
Nano Arduino" of the user manual of this DLL ("Universal DLL.pdf").
*/


/* XyzMoveToPosition(...) and XyzMoveToAngle(...) move, respectively, the linear position and the angle to
    the absolute values supplied in the arguments. Arguments are pointers to double[3] containing the new values.
 The routines are expected to return immediately - waiting for position is handled by OMDAQ
*/
XYZ_DLL bool _CALLSTYLE_ XyzMoveToPosition(double * NewPosition) {


    /*
    MF: In this function I just create a string with the motion order for the NanoArduino.
    For each available linear degree of freedom I convert the
    desired position into number of steps and check the
    direction of motion. I also add 0's in the string for the available angular degrees of freedom
    since the stage shouldn't move on these.
    */


  for (int i = 0; i < 3; ++i) {
    DemandPosition[i] = NewPosition[i];
  }



  vector<string> dir={};
  vector<string> nsteps={};
  string Stp;
  string order_aux;
  double nsteps_aux;



  for (int i = 0; i < 3; ++i) {



    if(Axis[i]) {



        if (NewPosition[i]>CurrentDllPosition[i]) {

            nsteps_aux = round((NewPosition[i]-CurrentDllPosition[i])/PosStep[i]);
            Stp=to_string(nsteps_aux);
            string::size_type k = Stp.find(".");
            Stp.erase(k, string::npos);
            nsteps.push_back(Stp);
            dir.push_back("1");

        }



        else {

            nsteps_aux = round((CurrentDllPosition[i]-NewPosition[i])/PosStep[i]);
            Stp=std::to_string(nsteps_aux);
            string::size_type k = Stp.find(".");
            Stp.erase(k, string::npos);
            nsteps.push_back(Stp);

            dir.push_back("0");
        }


    }



  }



  //MF: create order string. The number of steps and directions
  //for each available linear axis was already stored in vectors

  if(PosDOF==1){
    order_aux=dir[0]+" "+nsteps[0];
  }

  else if(PosDOF==2) {
    order_aux=dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1];
  }

  else if(PosDOF==3) {
    order_aux=dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+" "+dir[2]+" "+ nsteps[2];
  }



  if(AngleDOF==1){
    order_aux=order_aux+" 0 0";
  }

  else if(AngleDOF==2) {
    order_aux=order_aux+" 0 0 0 0";;
  }

  else if(AngleDOF==3) {
    order_aux=order_aux+" 0 0 0 0 0 0";
  }



  order_aux=order_aux+"\n";

  const char* order=order_aux.data();

```

```

  //MF: just to see the order string when a motion order is sent. For debbuging purposes.
  //To see it just set show_orders to true
  if(show_orders) {
  int msgboxID3 = MessageBoxA(
        NULL,
        order,
        "ORDER LINEAR MOTION",
        MB_ICONWARNING | MB_OK
    );
     }




  if(port){
  RS232_cputs(port_nmr, order);
   }




  tLin = clock();
  return true;
}







XYZ_DLL bool _CALLSTYLE_ XyzMoveToAngle(double * NewAngle) {


    /*MF:In this function I do exactly the same as I did the XyzMoveToPosition(...) but
    with the angular axis instead of the linear axis. I create a string with the motion
    order for the NanoArduino.For each available rotational degree of freedom I convert the
    desired position into number of steps and check the direction of motion. I also add 0's
    in the string for the available linear degrees of freedom since the stage shouldn't move on these
    */



  for (int i = 0; i < 3; ++i) {
    DemandAngle[i] = NewAngle[i];
  }

  vector<string> dir={};
  vector<string> nsteps={};
  string Stp;
  string order_aux;
  double nsteps_aux;






  for (int i = 0; i < 3; ++i) {


    if(Axis[i+3]) {



        if (NewAngle[i]>CurrentDllAngle[i]) {


            nsteps_aux = round((NewAngle[i]-CurrentDllAngle[i])/AngleStep[i]);
            Stp=to_string(nsteps_aux);
            string::size_type k = Stp.find(".");
            Stp.erase(k, string::npos);
            nsteps.push_back(Stp);
            dir.push_back("1");


        }



        else {


            nsteps_aux = round((CurrentDllAngle[i]-NewAngle[i])/AngleStep[i]);
            Stp=to_string(nsteps_aux);
            string::size_type k = Stp.find(".");
            Stp.erase(k, string::npos);
            nsteps.push_back(Stp);

            dir.push_back("0");


        }



    }



  }



  /*MF: create order string. The number of steps and directions
  for each available linear axis was already stored in vectors
  First I add 0's for each available linear axis and then
  I add the angular part*/


  if(PosDOF==1){
    order_aux="0 0 ";
  }
  else if (PosDOF==2) {
    order_aux="0 0 0 0 ";
  }
  else if (PosDOF==3) {
    order_aux="0 0 0 0 0 0 ";
  }





  if(AngleDOF==1){
    order_aux=order_aux+dir[0]+" "+nsteps[0]+"\n";
  }
  else if(AngleDOF==2) {
    order_aux=order_aux+dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+"\n";
  }
  else if(AngleDOF==3) {
```

```
    order_aux=order_aux+dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+" "+dir[2]+" "+ nsteps[2]+"\n";
  }




  const char* order=order_aux.data();

  if(port){
  RS232_cputs(port_nmr, order);
  }


  //MF: just to see the order string when a motion order is sent. For debbuging purposes.
  //To see it just set show_orders to true
  if(show_orders){
  int msgboxID3 = MessageBoxA(
        NULL,
        order,
        "ORDER ROT MOTION",
        MB_ICONWARNING | MB_OK
    );
   }




  tRot = clock();
  return true;
}

// XyzStop(...) is meant to perform an immediate halt (emergency stop, so no deceleration) on all axes
XYZ_DLL bool _CALLSTYLE_ XyzHalt() {


    //MF: this was not a needed funcionality
    //so I did nothing with this function

  return true;
}


/********************************** End of routines for motion command *******************************************/





/********************************** Routines for stage status reporting **********************************************/




/* XyzGetPosition(...) and XyzGetAngle(...) provide the current value of the linear and angular positions, respectively,
 to OMDAQ-3. This value is exported in the arguments, which are pointers to double[3].
*/
XYZ_DLL bool _CALLSTYLE_ XyzGetPosition(double * CurrentPosition) {
  clock_t tNow = clock();


  /*MF: In this function I just calculate the linear position of the stage
    Since stepping motors move in discrete steps I just check what is the closest
   approximation to the required position that the stage is capable of. (Note that
   the user can provide, through OMDAQ-3, any value for the demanded position)
  */





  double step_aux;




  for(int i=0; i<3; ++i) {



  if(Axis[i]){
    //MF: step slightly smaller than real step just because of the (finite) precision of doubles; to prevent unexpected behaviour


    step_aux=PosStep[i]-0.001*PosStep[i];




      if(DemandPosition[i]>CurrentDllPosition[i]){


      //MF: this "while" gets the closest approximation to the required position, by defect
        while ((DemandPosition[i]-CurrentDllPosition[i]) >= step_aux ) {
            CurrentDllPosition[i]+= PosStep[i];
      }


      //MF: after getting the closest approximation by defect, this "if" checks if adding one more
      //step would not result in a closer approximation to the required position
      if(abs(CurrentDllPosition[i]+PosStep[i] - DemandPosition[i])<abs(CurrentDllPosition[i] - DemandPosition[i])) {

          CurrentDllPosition[i]+=PosStep[i];
      }



    }






      else if(DemandPosition[i]<CurrentDllPosition[i]){


      //MF: this "while" gets the closest approximation to the required position, by excess
        while ((CurrentDllPosition[i] - DemandPosition[i]) >= step_aux ) {
            CurrentDllPosition[i]-= PosStep[i];
      }




      //MF: after getting the closest approximation by excess, this "if" checks if subtracting one more
```

```
      //step would not result in a closer approximation to the required position
      if(abs(CurrentDllPosition[i]-PosStep[i] -
      DemandPosition[i])<abs(CurrentDllPosition[i] - DemandPosition[i])) {

          CurrentDllPosition[i]-=PosStep[i];
      }




    }





    CurrentPosition[i]=CurrentDllPosition[i];


  }







  else if(!Axis[i]) {
      CurrentDllPosition[i]=0;
      CurrentPosition[i]=0;
  }


  }


  tLin = tNow;
  return true;
}



XYZ_DLL bool _CALLSTYLE_ XyzGetAngle(double * CurrentAngle) {
  clock_t tNow = clock();


   /*MF: In this function I just calculate the rotational position of the stage
    Since stepping motors move in discrete steps I just check what is the closest
   approximation to the required position that the stage is capable of. (Note that
   the user can provide, through OMDAQ-3, any value for the demanded position)
  */




  double step_aux;




  //MF: obtaining current motor position from the angle demanded by the user and the
  motor's precision

  for(int i=0; i<3; ++i) {


    if(Axis[i+3]){

    //MF: step slightly smaller than real step just because of the (finite)
    precision of doubles
        step_aux=AngleStep[i]-0.001*AngleStep[i];


        if(DemandAngle[i]>CurrentDllAngle[i]){

          //MF: this "while" gets the closest approximation to the required angle position, by defect
            while ((DemandAngle[i]-CurrentDllAngle[i]) >= step_aux ) {
                CurrentDllAngle[i]+= AngleStep[i];
            }

            //MF: after getting the closest approximation by defect, this "if" checks if adding one more
            //step would not result in a closer approximation to the required angle position
            if(abs(CurrentDllAngle[i]+AngleStep[i] - DemandAngle[i])<abs(CurrentDllAngle[i] - DemandAngle[i])) {

                CurrentDllAngle[i]+=AngleStep[i];
            }

        }



        else if(DemandAngle[i]<CurrentDllAngle[i]){
            //MF: this "while" gets the closest approximation to the required angle position, by excess
            while ((CurrentDllAngle[i] - DemandAngle[i]) >= step_aux ) {
                CurrentDllAngle[i]-= AngleStep[i];
            }

            //MF: after getting the closest approximation by excess, this "if" checks if subtracting one more
            //step would not result in a closer approximation to the required angle position
            if(abs(CurrentDllAngle[i]-AngleStep[i] - DemandAngle[i])<abs(CurrentDllAngle[i] - DemandAngle[i])) {

                CurrentDllAngle[i]-=AngleStep[i];
            }

        }



        CurrentAngle[i]=CurrentDllAngle[i];


    }



    else if(!Axis[i+3]) {
        CurrentDllAngle[i]=0;
        CurrentAngle[i]=0;
    }


  }



  tRot = tNow;
  return true;
}


```

```


/*
 GetMotorTemp is meant to return the temperature in degrees of all axes.
 MotorTemp is a pointer to a double array
 If iAxis = -1 MotorTemp is an array big enough to hold all motor temperatures.
 If iAxis >= 0 the temperature of iAxis is put into the first element of the array.
*/
XYZ_DLL bool _CALLSTYLE_ XyzGetMotorTemp(double *MotorTemp, int iAxis) {
  int iMin = 0;
  int iMax = 6;
  if (iAxis >= 0) {
    iMin = iAxis;
    iMax = iAxis;
  }
  for (int i = iMin; i < iMax; ++i) {
    int iDest = 0;
    if (iAxis >= 0) {
      iDest = i;
    }
    MotorTemp[iDest] = 25 + 0.01 * random(500);
  }
}




/* XyzStageStatus(...) informs OMDAQ-3 of what is happening with the stage by returning the
 DRVSTAT (UINT64) status mask built from the mask constants defined in OmXyzDll_StatusBits.h.
 Optionally the program may ask for more details in the AxisStatus DWORDs by passing a non-NULL
 pointer to AxisStatus.  This is DWORD[3] or DWORD[6] depending on the capabilities of the stage.
 If iAxis = -1 AxisStatus is an array big enough to hold all axis status.
 If iAxis >= 0 the status of iAxis is put into the first element of the array.
 Note that for single axis calls only the single axis segments of status are filled
 so this must be managed in the calling program.
*/
XYZ_DLL DRVSTAT _CALLSTYLE_ XyzStageStatus(DWORD * AxisStatus) {
  return XyzAxisStatus(-1, AxisStatus);
}

XYZ_DLL DRVSTAT _CALLSTYLE_ XyzAxisStatus(int iAxis, DWORD * AxisStatus) {
  DRVSTAT status = 0;
  int iMin = 0;
  int iMax = 6;

  if (iAxis >= 0) {
    iMin = iAxis;
    iMax = iAxis;
  }

  /*MF: Removed all the verifications if the stage had reached any hardware limits.
  This is not strictly necessary. Check if each axis is available or not for the stage in use.
  If it is available check if the stage is in motion or not, along that axis,
  and set flags accordingly.
  */

  double step_aux;

  for (int i = iMin; i < iMax; ++i) {

    if (Axis[i]) {



        if(i<3) {

            step_aux=PosStep[i]-0.001*PosStep[i];

            if (fabs(CurrentDllPosition[i] - DemandPosition[i]) >= step_aux) {

                switch (i) {
                    case 0:
                        status = status | ST_AX1_MOVING;
                        break;
                    case 1:
                        status = status | ST_AX2_MOVING;
                        break;
                    case 2:
                        status = status | ST_AX3_MOVING;
                        break;



                }
            }

            else {

                switch (i) {
                    case 0:
                        status = status | ST_AX1_INPOSITION;
                        break;
                    case 1:
                        status = status | ST_AX2_INPOSITION;
                        break;
                    case 2:
                        status = status | ST_AX3_INPOSITION;
                        break;

                }
            }

        }



        else {

            step_aux=AngleStep[i-3]-0.001*AngleStep[i-3];

            if (fabs(CurrentDllAngle[i-3] - DemandAngle[i-3]) >= step_aux) {

                switch (i) {
                    case 3:
                        status = status | ST_RO1_MOVING;
                        break;
                    case 4:
                        status = status | ST_RO2_MOVING;
                        break;
                    case 5:
                        status = status | ST_RO3_MOVING;
                        break;

                }

            }


            else {

                switch (i) {
                    case 3:
                        status = status | ST_RO1_INPOSITION;
                        break;
                    case 4:
                        status = status | ST_RO2_INPOSITION;
                        break;
                    case 5:
                        status = status | ST_RO3_INPOSITION;
                        break;
                }

            }

        }

    }



    else if(!Axis[i]){

        switch (i) {
            case 0:
                status = status | ST_AX1_INPOSITION;
                break;
            case 1:
                status = status | ST_AX2_INPOSITION;
                break;
            case 2:
                status = status | ST_AX3_INPOSITION;
                break;
        }
    }




  }



  //MF: According to the degrees of freedom available for the stage, flags are set to inform that the rotational motors are on
  if (AngleDOF==1) {
    status = status | ST_RO1_MOTOR_ON;
  }
  else if(AngleDOF==2) {
    status = status | ST_RO1_MOTOR_ON | ST_RO2_MOTOR_ON;
  }
  else if(AngleDOF==3) {
    status = status | ST_RO1_MOTOR_ON | ST_RO2_MOTOR_ON | ST_RO3_MOTOR_ON;
  }

  //MF: The linear motors always "on". OMDAQ-3 requires that the stage have linear translation in the 3 cartesian axes. Even if
  //the stage is not capable of this OMDAQ-3 is still informed otherwise
  if (DllPowerOn) {
    status |= (ST_ALL_XYZ_MOTORS_ON );
  }



  return status;


}


/********************************** End of routines for stage status reporting **********************************************/

```

```
/*********************************** Routines for handling errors **********************************************/

/*
 XyzFaultAck() is called after StageStatus reports a fault - defined with the mask constants with the
 terminations "POSLIM", "NEGLIM" or "HWFAULT" on any axis.  This should be used to clear faults
 (e.g. backing off from limit switches).
 Return values have the followinhg meanings:
 XyzFltAckOK    0    - Fault has been cleared OK (as far as I can tell)
 XyzFltAckFatal 1    - Fault cannot be cleared and the stage is dead
 (in which case OMDAQ will try to do a tidy shutdown)
 XyzFtlAckRetry 2    - I may be able to clear the fault if you try again,
*/
XYZ_DLL int _CALLSTYLE_ XyzFaultAck() {


    //MF: this was not a needed funcionality
    //so I did nothing with this function

  return XyzFltAckOK;
}

/*
 XyzLastFaultText(...) call returns a text description of the last HWFAULT encountered
 The existence of a fault must be signalled in the StageStatus flag mask.
 nChar is the length of the supplied buffer.    (typically 80 characters)

 return true for success.
*/
XYZ_DLL bool _CALLSTYLE_ XyzLastFaultText(char *statusText, int nChar) {

  //MF: this was not a needed funcionality
  //so I did nothing with this function

  strcpy(statusText, "Fault?  What fault?");


  return true;
}




/*********************************** End of routines for handling errors **********************************************/
```