```
// ---------------------------------------------------------------------------
//
// 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;

```

```
64  //MF: Global variables for debbuging and testing purposes
65  bool port=true; /*MF: variable used to prevent RS-232 communications just to test
    the DLL without the hardware
66  if false no RS232 orders are sent and there's no error when linking OMDAQ-3 with the
    DLL
67  without the actual RS232 connection established */
68  bool show_orders=false; /*MF: variable used to show the motion orders in a pop up
    window. When set to true
69  the strings sent to the NanoArduino appear in a pop up window.*/
70
71  using namespace std;
72
73
74
75  //MF: added function
76  //MF: function to check if char array has only numbers
77  //MF: used to check if parameters given in the OMDAQ-3 parameters window are okay
78  bool ISnumber(char *c) {
79      string c_s(c);
80      bool has_only_digits = (c_s.find_first_not_of( "0123456789." ) == string::npos);
81      return has_only_digits;
82  };
83
84
85
86
87
88  /****************************** Adminstration routines
    ******************************/
89
90
91
92
93  /*XyzCapabilityMask returns a DWORD mask that describes the basic functionality
94   of the hardware and allows OMDAQ to make the user interface.
95   The return value is assembled from the capability constants
96   defined in OmXyzDll_StatusBits.h */
97   /*>>>>>> THIS MUST BE DEFINED <<<<<<<*/
98  XYZ_DLL DWORD _CALLSTYLE_ XyzCapabilityMask() {
99      unsigned long CAP=0;
100     //MF: motion along the 3 cartesian axis must always be declared (even if the
        stage is not able to perform such motion)
101     CAP= CAP | XYZCAP_XYZ3;
102
103     /*MF: When OMDAQ-3 is executed there is a window that pops up asking for
        parameters. The user inputs the parameters to
104      set up the RS232 communication channel and provides the step for each available
         axis of the stage.
105      After the parameters have been provided and the pop up window is closed the
         initialisation function
106      XyzInitialise(char **options, int szOptions) is called and the parameters are
         retrieved through the char **options argument.
107      This function - XyzCapabilityMask() - is called, for some reason, before and
         after the initialisation function. In the case
108      of this stage in particular the available axes are only known after the
         options have been retrieved, i.e., copied, which is
109      indicated by the bool optionsCopied.*/
110
111     //MF: if optionsCopied==true then the initialize function was already called and
        we already know the number of degrees of freedom
112     if(optionsCopied) {
113
114         if(AngleDOF==1) {
115             CAP= CAP | XYZCAP_ROT1;
116         }
117         else if(AngleDOF==2) {
118             CAP= CAP | XYZCAP_ROT2;
119         }
120         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 ******************************/

```

```
252
253 // ---------------------------------------------------------------------------
254 XYZ_DLL bool _CALLSTYLE_ XyzInitialise(char **options, int szOptions) {
255   /* Initialisation code here
256
257    This function obtains the parameter values from the parameters window interface
258    through the char **options argument. Namely the RS232 communication parameters
259    and the step of each avaliable axis. Global variables are defined with these
       parameters.
260
261    The RS232 channel is opened.
262
263    */
264
265
266
267    /*MF: added a pop up warning that is launched when OMDAQ-3 is executed
268    so that the user is careful when setting up the parameters of the stage.
269    Wrong steps can result in stage damage if hardware limits are breached */
270    int msgboxID = MessageBoxA(
271            NULL,
272            "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.",
273            "WARNING - motorized stage",
274            MB_TOPMOST | MB_ICONWARNING | MB_OK
275        );
276
277
278    if (szOptions != nOptions) {
279        return false;
280    }
281
282
283    for (int i = 0; i < szOptions; ++i) {
284        ZeroMemory(&OptionText[i][0], 32*nOptions*sizeof(char));
285        strcpy(&OptionText[i][0], options[i]);
286    }
287    optionsCopied = true;
288
289
290
291
292
293    for (int i = 0; i < 3; ++i) {
294        CurrentDllPosition[i] = 0;
295        CurrentDllAngle[i] = 0;
296    }
297
298
299
300
301
302
303    //MF: checking the used axis according to the parameters given in the parameters
       window interface
304    //MF: setting variables bool Axis[6] with the used axes and int PosDOF with the
       number of linear degrees of freedom
305    double LinSteps[3];
306    double RotSteps[3];
```

```
307       PosDOF=0;
308       for (int i=0; i < 3; i++) {
309           if(ISnumber(options[3+i])){
310               PosStep[i]=atof(options[3+i]);
311               PosDOF+=1;
312               Axis[i]=true;
313           }
314           else {
315               PosStep[i]=0;
316               Axis[i]=false;
317           }
318       }
319
320
321        //MF: setting variables bool Axis[6] with the used axes and int AngleDOF with
           the number of rotational degrees of freedom
322       AngleDOF=0;
323       int k=0;
324       for (int i = 0; i < 3; i++) {
325             if(ISnumber(options[6+i])) {
326               AngleStep[k]=atof(options[6+i]);
327               AngleDOF+=1;
328               Axis[k+3]=true;
329               k=k+1;
330           }
331       }
332
333       for (int i=k; i<3; i++){
334           AngleStep[i]=0;
335           Axis[i+3]=false;
336       }
337
338
339
340
341     //MF: opening COM port
342     //MF: getting RS232 parameters from OMDAQ-3 parameters window
343
344       if(options[0]!=NULL){
345       port_nmr=atoi(options[0])-1;
346       }
347
348       if(options[1]!=NULL){
349       taxabaud=atoi(options[1]);
350       }
351
352       if(options[2]!=NULL){
353       std::strcpy(modo,options[2]);
354       }
355
356
357     /*MF: openning communications port. If something goes wrong
358     a pop up message appears saying that there was an error */
359       if(port){
360
361
362           if(RS232_OpenComport(port_nmr, taxabaud, modo))    {
363
364               int msgboxID3 = MessageBoxA(
365                   NULL,
366                   "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",
367                   "ERROR",
368                   MB_TOPMOST | MB_ICONERROR | MB_OK
369               );
370
```

```
371
372             return(0);
373
374             }
375
376
377         }
378
379
380
381         DllPowerOn = true;
382         return true;
383
384     }
385
386     /*-------------------------------------------------------------------------*/
387     XYZ_DLL bool _CALLSTYLE_ XyzShutDown() {
388       /* Full shutdown code here  - stop stage if it's moving,
389        power down, free comms links and free resources.
390
391        OMDAQ saves the position at shutdown ready for the next startup.
392        return false if it fails. */
393       return true;
394     }
395
396     /*-------------------------------------------------------------------------*/
397     /* This procedure initialises the values of the position readouts to the supplied
       values.
398      NewPosition is a pointer to a double[3] array which contains the
399      new values of the absolute postions for axes 0..2 */
400     XYZ_DLL bool _CALLSTYLE_ XyzSetCurrentPosition(double * NewPosition) {
401
402       /*MF: if a linear axis is available the linear position variables are updated with
         whatever value
403       intended. If not the linear position variables are set to 0 */
404       for (int i = 0; i < 3; ++i) {
405         if(Axis[i]){
406             CurrentDllPosition[i] = NewPosition[i];
407             DemandPosition[i] = NewPosition[i];   }
408         else if(!Axis[i]){
409             CurrentDllPosition[i]=0;
410             DemandPosition[i]=0;  }
411
412       }
413
414       return true;
415     }
416
417
418     /* This procedure initialises the values of the angle readouts to the supplied values.
419     NewAngle is a pointer to a double[3] array which contains the
420      new values of the absolute angles for axes 3..5 */
421     XYZ_DLL bool _CALLSTYLE_ XyzSetCurrentAngle(double * NewAngle) {
422
423       /*MF: if a rotational axis is available the angular position variables are updated
         with whatever value
424       intended. If not the angular position variables are set to 0 */
425       for (int i = 0; i < 3; ++i) {
426         if(Axis[i+3]){
427             CurrentDllAngle[i] = NewAngle[i];
428             DemandAngle[i] = NewAngle[i];   }
429         else if(!Axis[i+3]){
430             CurrentDllAngle[i]=0;
431             DemandAngle[i]=0;  }
432
433
434       }
435       return true;
436     }
```

```
437
438 /********************************* End of initialisation routines
    ******************************/
439
440
441
442
443
444
445 /********************************* Routines to set up motion parameters
    ******************************/
446
447 /* XyzSetSpeed(...) and XyzSetAccel(...) set the LINEAR speed and acceleration per
    axis,
448 respectively. The acceleration is assumed to be the same in the accel and decel
    phases.  Units are  mm/sec and mm/sec2.
449 NewAccel and NewSpeed are pointers to double[3] arrays containing the new values for
    each axis.
450 At present OMDAQ only allows a single accel value for all axes. */
451 XYZ_DLL bool _CALLSTYLE_ XyzSetAccel(double * NewAccel) {
452     /*This was not a needed funcionality
453     so this function was not used. */
454   return true;
455 }
456
457 XYZ_DLL bool _CALLSTYLE_ XyzSetSpeed(double * NewSpeed) {
458
459     /*This was not a needed funcionality
460     so this function was not used.*/
461
462   for (int i = 0; i < 3; ++i) {
463     LinSpeed[i] = NewSpeed[i];
464   }
465
466   return true;
467 }
468
469 /* XyzSetRotSpeed(...) and XyzSetRotAccel set the ROTATIONAL speed and acceleration
    per axis, respectively.
470  The acceleration is assumed to be the same in the accel and decel phases.  Units
     are  deg/sec and deg/sec2
471  NewAccel and NewSpeed are pointers to double[3] arrays containing the new values
     for each axis.   */
472 XYZ_DLL bool _CALLSTYLE_ XyzSetRotAccel(double * NewAccel) {
473
474     /*This was not a needed funcionality
475     so this function was not used.*/
476
477   return true;
478 }
479
480 XYZ_DLL bool _CALLSTYLE_ XyzSetRotSpeed(double * NewSpeed) {
481
482     /*This was not a needed funcionality
483     so this function was not used.*/
484
485   for (int i = 0; i < 3; ++i) {
486     RotSpeed[i] = NewSpeed[i];
487   }
488   return true;
489 }
490
491 /* XyzPowerOn(...) is meant to control the power of the stage (on or off).
492  If Enabled = true (false) the power should be turned ON (OFF) for all axes.
493  It should leave the controller active and reporting.
494  Returns true for success. */
495 XYZ_DLL bool _CALLSTYLE_ XyzPowerOn(bool Enabled) {
496
497     /*This was not a needed funcionality
```

```
498       so this function was not used.*/
499
500     DllPowerOn = Enabled;
501     return true;
502   }
503
504
505
506   /********************************* End of routines to set up motion parameters
      ******************************/
507
508
509
510
511
512
513
514   /********************************* Routines for motion command
      ****************************************/
515
516
517   /*MF: To understand the format of the strings sent to the Arduino Nano please read
      section 2 . "Communication with
518   Nano Arduino" of the user manual of this DLL ("Universal DLL.pdf").
519   */
520
521
522   /* XyzMoveToPosition(...) and XyzMoveToAngle(...) move, respectively, the linear
      position and the angle to
523       the absolute values supplied in the arguments. Arguments are pointers to
          double[3] containing the new values.
524    The routines are expected to return immediately - waiting for position is handled
       by OMDAQ
525   */
526   XYZ_DLL bool _CALLSTYLE_ XyzMoveToPosition(double * NewPosition) {
527
528
529       /*
530       MF: In this function I just create a string with the motion order for the
          NanoArduino.
531       For each available linear degree of freedom I convert the
532       desired position into number of steps and check the
533       direction of motion. I also add 0's in the string for the available angular
          degrees of freedom
534       since the stage shouldn't move on these.
535       */
536
537
538     for (int i = 0; i < 3; ++i) {
539       DemandPosition[i] = NewPosition[i];
540     }
541
542
543
544     vector<string> dir={};
545     vector<string> nsteps={};
546     string Stp;
547     string order_aux;
548     double nsteps_aux;
549
550
551
552     for (int i = 0; i < 3; ++i) {
553
554
555
556       if(Axis[i]) {
557
558
```

```
559
560         if (NewPosition[i]>CurrentDllPosition[i]) {
561
562             nsteps_aux = round((NewPosition[i]-CurrentDllPosition[i])/PosStep[i]);
563             Stp=to_string(nsteps_aux);
564             string::size_type k = Stp.find(".");
565             Stp.erase(k, string::npos);
566             nsteps.push_back(Stp);
567             dir.push_back("1");
568
569         }
570
571
572
573         else {
574
575             nsteps_aux = round((CurrentDllPosition[i]-NewPosition[i])/PosStep[i]);
576             Stp=std::to_string(nsteps_aux);
577             string::size_type k = Stp.find(".");
578             Stp.erase(k, string::npos);
579             nsteps.push_back(Stp);
580
581             dir.push_back("0");
582         }
583
584
585       }
586
587
588
589     }
590
591
592
593     //MF: create order string. The number of steps and directions
594     //for each available linear axis was already stored in vectors
595
596     if(PosDOF==1){
597       order_aux=dir[0]+" "+nsteps[0];
598     }
599
600     else if(PosDOF==2) {
601       order_aux=dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1];
602     }
603
604     else if(PosDOF==3) {
605       order_aux=dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+" "+dir[2]+" "+ nsteps[2];
606     }
607
608
609
610     if(AngleDOF==1){
611       order_aux=order_aux+" 0 0";
612     }
613
614     else if(AngleDOF==2) {
615       order_aux=order_aux+" 0 0 0 0";;
616     }
617
618     else if(AngleDOF==3) {
619       order_aux=order_aux+" 0 0 0 0 0 0";
620     }
621
622
623
624     order_aux=order_aux+"\n";
625
626     const char* order=order_aux.data();
627
```

```
628
629     //MF: just to see the order string when a motion order is sent. For debbuging
        purposes.
630     //To see it just set show_orders to true
631     if(show_orders) {
632     int msgboxID3 = MessageBoxA(
633           NULL,
634           order,
635           "ORDER LINEAR MOTION",
636           MB_ICONWARNING | MB_OK
637       );
638        }
639
640
641
642
643     if(port){
644     RS232_cputs(port_nmr, order);
645      }
646
647
648
649
650     tLin = clock();
651     return true;
652   }
653
654
655
656
657
658
659
660   XYZ_DLL bool _CALLSTYLE_ XyzMoveToAngle(double * NewAngle) {
661
662
663       /*MF:In this function I do exactly the same as I did the XyzMoveToPosition(...)
          but
664       with the angular axis instead of the linear axis. I create a string with the
          motion
665       order for the NanoArduino.For each available rotational degree of freedom I
          convert the
666       desired position into number of steps and check the direction of motion. I also
          add 0's
667       in the string for the available linear degrees of freedom since the stage
          shouldn't move on these
668       */
669
670
671
672     for (int i = 0; i < 3; ++i) {
673       DemandAngle[i] = NewAngle[i];
674     }
675
676     vector<string> dir={};
677     vector<string> nsteps={};
678     string Stp;
679     string order_aux;
680     double nsteps_aux;
681
682
683
684
685
686
687     for (int i = 0; i < 3; ++i) {
688
689
690       if(Axis[i+3]) {
```

```
691
692
693
694         if (NewAngle[i]>CurrentDllAngle[i]) {
695
696
697             nsteps_aux = round((NewAngle[i]-CurrentDllAngle[i])/AngleStep[i]);
698             Stp=to_string(nsteps_aux);
699             string::size_type k = Stp.find(".");
700             Stp.erase(k, string::npos);
701             nsteps.push_back(Stp);
702             dir.push_back("1");
703
704
705         }
706
707
708
709         else {
710
711
712             nsteps_aux = round((CurrentDllAngle[i]-NewAngle[i])/AngleStep[i]);
713             Stp=to_string(nsteps_aux);
714             string::size_type k = Stp.find(".");
715             Stp.erase(k, string::npos);
716             nsteps.push_back(Stp);
717
718             dir.push_back("0");
719
720
721         }
722
723
724
725       }
726
727
728
729     }
730
731
732
733     /*MF: create order string. The number of steps and directions
734     for each available linear axis was already stored in vectors
735     First I add 0's for each available linear axis and then
736     I add the angular part*/
737
738
739     if(PosDOF==1){
740       order_aux="0 0 ";
741     }
742     else if (PosDOF==2) {
743       order_aux="0 0 0 0 ";
744     }
745     else if (PosDOF==3) {
746       order_aux="0 0 0 0 0 0 ";
747     }
748
749
750
751
752
753     if(AngleDOF==1){
754       order_aux=order_aux+dir[0]+" "+nsteps[0]+"\n";
755     }
756     else if(AngleDOF==2) {
757       order_aux=order_aux+dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+"\n";
758     }
759     else if(AngleDOF==3) {
```

```
760       order_aux=order_aux+dir[0]+" "+nsteps[0]+" "+dir[1]+" "+nsteps[1]+" "+dir[2]+"
          "+ nsteps[2]+"\n";
761     }
762
763
764
765
766
767     const char* order=order_aux.data();
768
769     if(port){
770     RS232_cputs(port_nmr, order);
771     }
772
773
774     //MF: just to see the order string when a motion order is sent. For debbuging
        purposes.
775     //To see it just set show_orders to true
776     if(show_orders){
777     int msgboxID3 = MessageBoxA(
778           NULL,
779           order,
780           "ORDER ROT MOTION",
781           MB_ICONWARNING | MB_OK
782       );
783      }
784
785
786
787
788     tRot = clock();
789     return true;
790   }
791
792   // XyzStop(...) is meant to perform an immediate halt (emergency stop, so no
      deceleration) on all axes
793   XYZ_DLL bool _CALLSTYLE_ XyzHalt() {
794
795
796       //MF: this was not a needed funcionality
797       //so I did nothing with this function
798
799     return true;
800   }
801
802
803   /********************************** End of routines for motion command
      ****************************************/
804
805
806
807
808
809   /********************************** Routines for stage status reporting
      ********************************************/
810
811
812
813
814   /* XyzGetPosition(...) and XyzGetAngle(...) provide the current value of the linear
      and angular positions, respectively,
815    to OMDAQ-3. This value is exported in the arguments, which are pointers to double[3].
816   */
817   XYZ_DLL bool _CALLSTYLE_ XyzGetPosition(double * CurrentPosition) {
818     clock_t tNow = clock();
819
820
821     /*MF: In this function I just calculate the linear position of the stage
822       Since stepping motors move in discrete steps I just check what is the closest
```

```
823      approximation to the required position that the stage is capable of. (Note that
824      the user can provide, through OMDAQ-3, any value for the demanded position)
825     */
826
827
828
829
830
831     double step_aux;
832
833
834
835
836     for(int i=0; i<3; ++i) {
837
838
839
840     if(Axis[i]){
841       //MF: step slightly smaller than real step just because of the (finite)
          precision of doubles; to prevent unexpected behaviour
842
843
844       step_aux=PosStep[i]-0.001*PosStep[i];
845
846
847
848
849         if(DemandPosition[i]>CurrentDllPosition[i]){
850
851
852         //MF: this "while" gets the closest approximation to the required position, by
            defect
853           while ((DemandPosition[i]-CurrentDllPosition[i]) >= step_aux ) {
854               CurrentDllPosition[i]+= PosStep[i];
855         }
856
857
858         //MF: after getting the closest approximation by defect, this "if" checks if
            adding one more
859         //step would not result in a closer approximation to the required position
860         if(abs(CurrentDllPosition[i]+PosStep[i] -
            DemandPosition[i])<abs(CurrentDllPosition[i] - DemandPosition[i])) {
861
862             CurrentDllPosition[i]+=PosStep[i];
863         }
864
865
866
867       }
868
869
870
871
872
873
874         else if(DemandPosition[i]<CurrentDllPosition[i]){
875
876
877         //MF: this "while" gets the closest approximation to the required position, by
            excess
878           while ((CurrentDllPosition[i] - DemandPosition[i]) >= step_aux ) {
879               CurrentDllPosition[i]-= PosStep[i];
880         }
881
882
883
884
885         //MF: after getting the closest approximation by excess, this "if" checks if
            subtracting one more
```

```
886         //step would not result in a closer approximation to the required position
887         if(abs(CurrentDllPosition[i]-PosStep[i] -
            DemandPosition[i])<abs(CurrentDllPosition[i] - DemandPosition[i])) {
888
889             CurrentDllPosition[i]-=PosStep[i];
890         }
891
892
893
894
895       }
896
897
898
899
900
901       CurrentPosition[i]=CurrentDllPosition[i];
902
903
904     }
905
906
907
908
909
910
911
912     else if(!Axis[i]) {
913         CurrentDllPosition[i]=0;
914         CurrentPosition[i]=0;
915     }
916
917
918     }
919
920
921     tLin = tNow;
922     return true;
923   }
924
925
926
927   XYZ_DLL bool _CALLSTYLE_ XyzGetAngle(double * CurrentAngle) {
928     clock_t tNow = clock();
929
930
931      /*MF: In this function I just calculate the rotational position of the stage
932       Since stepping motors move in discrete steps I just check what is the closest
933      approximation to the required position that the stage is capable of. (Note that
934      the user can provide, through OMDAQ-3, any value for the demanded position)
935     */
936
937
938
939
940     double step_aux;
941
942
943
944
945     //MF: obtaining current motor position from the angle demanded by the user and the
        motor's precision
946
947     for(int i=0; i<3; ++i) {
948
949
950       if(Axis[i+3]){
951
952       //MF: step slightly smaller than real step just because of the (finite)
```

```
      precision of doubles
953         step_aux=AngleStep[i]-0.001*AngleStep[i];
954
955
956         if(DemandAngle[i]>CurrentDllAngle[i]){
957
958           //MF: this "while" gets the closest approximation to the required angle
              position, by defect
959             while ((DemandAngle[i]-CurrentDllAngle[i]) >= step_aux ) {
960                 CurrentDllAngle[i]+= AngleStep[i];
961             }
962
963             //MF: after getting the closest approximation by defect, this "if"
                checks if adding one more
964             //step would not result in a closer approximation to the required angle
                position
965             if(abs(CurrentDllAngle[i]+AngleStep[i] -
                DemandAngle[i])<abs(CurrentDllAngle[i] - DemandAngle[i])) {
966
967                 CurrentDllAngle[i]+=AngleStep[i];
968             }
969
970         }
971
972
973
974         else if(DemandAngle[i]<CurrentDllAngle[i]){
975             //MF: this "while" gets the closest approximation to the required angle
                position, by excess
976             while ((CurrentDllAngle[i] - DemandAngle[i]) >= step_aux ) {
977                 CurrentDllAngle[i]-= AngleStep[i];
978             }
979
980             //MF: after getting the closest approximation by excess, this "if"
                checks if subtracting one more
981             //step would not result in a closer approximation to the required angle
                position
982             if(abs(CurrentDllAngle[i]-AngleStep[i] -
                DemandAngle[i])<abs(CurrentDllAngle[i] - DemandAngle[i])) {
983
984                 CurrentDllAngle[i]-=AngleStep[i];
985             }
986
987         }
988
989
990
991         CurrentAngle[i]=CurrentDllAngle[i];
992
993
994       }
995
996
997
998       else if(!Axis[i+3]) {
999         CurrentDllAngle[i]=0;
1000        CurrentAngle[i]=0;
1001      }
1002
1003
1004    }
1005
1006
1007
1008    tRot = tNow;
1009    return true;
1010  }
1011
1012
```

```
1013
1014
1015  /*
1016   GetMotorTemp is meant to return the temperature in degrees of all axes.
1017   MotorTemp is a pointer to a double array
1018   If iAxis = -1 MotorTemp is an array big enough to hold all motor temperatures.
1019   If iAxis >= 0 the temperature of iAxis is put into the first element of the array.
1020  */
1021  XYZ_DLL bool _CALLSTYLE_ XyzGetMotorTemp(double *MotorTemp, int iAxis) {
1022    int iMin = 0;
1023    int iMax = 6;
1024    if (iAxis >= 0) {
1025      iMin = iAxis;
1026      iMax = iAxis;
1027    }
1028    for (int i = iMin; i < iMax; ++i) {
1029      int iDest = 0;
1030      if (iAxis >= 0) {
1031        iDest = i;
1032      }
1033      MotorTemp[iDest] = 25 + 0.01 * random(500);
1034    }
1035  }
1036
1037
1038
1039
1040  /* XyzStageStatus(...) informs OMDAQ-3 of what is happening with the stage by
     returning the
1041   DRVSTAT (UINT64) status mask built from the mask constants defined in
     OmXyzDll_StatusBits.h.
1042   Optionally the program may ask for more details in the AxisStatus DWORDs by passing
     a non-NULL
1043   pointer to AxisStatus.  This is DWORD[3] or DWORD[6] depending on the capabilities
     of the stage.
1044   If iAxis = -1 AxisStatus is an array big enough to hold all axis status.
1045   If iAxis >= 0 the status of iAxis is put into the first element of the array.
1046   Note that for single axis calls only the single axis segments of status are filled
1047   so this must be managed in the calling program.
1048  */
1049  XYZ_DLL DRVSTAT _CALLSTYLE_ XyzStageStatus(DWORD * AxisStatus) {
1050    return XyzAxisStatus(-1, AxisStatus);
1051  }
1052
1053  XYZ_DLL DRVSTAT _CALLSTYLE_ XyzAxisStatus(int iAxis, DWORD * AxisStatus) {
1054    DRVSTAT status = 0;
1055    int iMin = 0;
1056    int iMax = 6;
1057
1058    if (iAxis >= 0) {
1059      iMin = iAxis;
1060      iMax = iAxis;
1061    }
1062
1063    /*MF: Removed all the verifications if the stage had reached any hardware limits.
1064    This is not strictly necessary. Check if each axis is available or not for the
     stage in use.
1065    If it is available check if the stage is in motion or not, along that axis,
1066    and set flags accordingly.
1067    */
1068
1069    double step_aux;
1070
1071    for (int i = iMin; i < iMax; ++i) {
1072
1073      if (Axis[i]) {
1074
1075
1076
```

```
1077            if(i<3) {
1078
1079                step_aux=PosStep[i]-0.001*PosStep[i];
1080
1081                if (fabs(CurrentDllPosition[i] - DemandPosition[i]) >= step_aux) {
1082
1083                    switch (i) {
1084                        case 0:
1085                            status = status | ST_AX1_MOVING;
1086                            break;
1087                        case 1:
1088                            status = status | ST_AX2_MOVING;
1089                            break;
1090                        case 2:
1091                            status = status | ST_AX3_MOVING;
1092                            break;
1093
1094
1095
1096                    }
1097                }
1098
1099                else {
1100
1101                    switch (i) {
1102                        case 0:
1103                            status = status | ST_AX1_INPOSITION;
1104                            break;
1105                        case 1:
1106                            status = status | ST_AX2_INPOSITION;
1107                            break;
1108                        case 2:
1109                            status = status | ST_AX3_INPOSITION;
1110                            break;
1111
1112                    }
1113                }
1114
1115            }
1116
1117
1118
1119            else {
1120
1121                step_aux=AngleStep[i-3]-0.001*AngleStep[i-3];
1122
1123                if (fabs(CurrentDllAngle[i-3] - DemandAngle[i-3]) >= step_aux) {
1124
1125                    switch (i) {
1126                        case 3:
1127                            status = status | ST_RO1_MOVING;
1128                            break;
1129                        case 4:
1130                            status = status | ST_RO2_MOVING;
1131                            break;
1132                        case 5:
1133                            status = status | ST_RO3_MOVING;
1134                            break;
1135
1136                    }
1137
1138                }
1139
1140
1141                else {
1142
1143                    switch (i) {
1144                        case 3:
1145                            status = status | ST_RO1_INPOSITION;
```

```
1146                        break;
1147                    case 4:
1148                        status = status | ST_RO2_INPOSITION;
1149                        break;
1150                    case 5:
1151                        status = status | ST_RO3_INPOSITION;
1152                        break;
1153                }
1154
1155            }
1156
1157        }
1158
1159    }
1160
1161
1162
1163    else if(!Axis[i]){
1164
1165        switch (i) {
1166            case 0:
1167                status = status | ST_AX1_INPOSITION;
1168                break;
1169            case 1:
1170                status = status | ST_AX2_INPOSITION;
1171                break;
1172            case 2:
1173                status = status | ST_AX3_INPOSITION;
1174                break;
1175        }
1176    }
1177
1178
1179
1180
1181  }
1182
1183
1184
1185  //MF: According to the degrees of freedom available for the stage, flags are set
      to inform that the rotational motors are on
1186  if (AngleDOF==1) {
1187    status = status | ST_RO1_MOTOR_ON;
1188  }
1189  else if(AngleDOF==2) {
1190    status = status | ST_RO1_MOTOR_ON | ST_RO2_MOTOR_ON;
1191  }
1192  else if(AngleDOF==3) {
1193    status = status | ST_RO1_MOTOR_ON | ST_RO2_MOTOR_ON | ST_RO3_MOTOR_ON;
1194  }
1195
1196  //MF: The linear motors always "on". OMDAQ-3 requires that the stage have linear
      translation in the 3 cartesian axes. Even if
1197  //the stage is not capable of this OMDAQ-3 is still informed otherwise
1198  if (DllPowerOn) {
1199    status |= (ST_ALL_XYZ_MOTORS_ON );
1200  }
1201
1202
1203
1204  return status;
1205
1206
1207}
1208
1209
1210/********************************** End of routines for stage status reporting
      ********************************************/
1211
```

```
1212
1213
1214
1215
1216  /******************************** Routines for handling errors
      ******************************************/
1217
1218  /*
1219   XyzFaultAck() is called after StageStatus reports a fault - defined with the mask
      constants with the
1220   terminations "POSLIM", "NEGLIM" or "HWFAULT" on any axis.  This should be used to
      clear faults
1221   (e.g. backing off from limit switches).
1222   Return values have the followinhg meanings:
1223   XyzFltAckOK    0    - Fault has been cleared OK (as far as I can tell)
1224   XyzFltAckFatal 1    - Fault cannot be cleared and the stage is dead
1225   (in which case OMDAQ will try to do a tidy shutdown)
1226   XyzFtlAckRetry 2    - I may be able to clear the fault if you try again,
1227  */
1228  XYZ_DLL int _CALLSTYLE_ XyzFaultAck() {
1229
1230
1231      //MF: this was not a needed funcionality
1232      //so I did nothing with this function
1233
1234    return XyzFltAckOK;
1235  }
1236
1237  /*
1238   XyzLastFaultText(...) call returns a text description of the last HWFAULT encountered
1239   The existence of a fault must be signalled in the StageStatus flag mask.
1240   nChar is the length of the supplied buffer.    (typically 80 characters)
1241
1242   return true for success.
1243  */
1244  XYZ_DLL bool _CALLSTYLE_ XyzLastFaultText(char *statusText, int nChar) {
1245
1246    //MF: this was not a needed funcionality
1247    //so I did nothing with this function
1248
1249    strcpy(statusText, "Fault?  What fault?");
1250
1251
1252    return true;
1253  }
1254
1255
1256
1257
1258  /******************************** End of routines for handling errors
      ******************************************/
1259
```