MarkerLibrary.h

Marker Library exported function definitions:

1. LoadMarkerLibrary:

Create a new MarkerLibraryRT instance, start the main object.

1. InitializeMarkerLibrary:

1) Reset the LastError code to be 0.

2) Create the Configuration Manager which will store the path of executing folder, PCLens, and PCLaser in the configuration manager and create all the other configure instances in the Configuration Manager.

3) Start the logging.

4) Initialize Winsock.

5) Create the JobManager.

6) Create the ConnectivityManager.

7) Create the Networking object.

8) Search for all the config files on the PC.

9) Get the license on the PC and this will create the right license in the license manager.

10) Cache the stroke fonts in the memory and start the object import filters.

11) Create the MotionManager object.

12) Create the HostManagerRT.

13) Initialize the HostManager.

14) Create the double buffer list controller.

15) Create the LECDetectionManager.

1. ReleaseLibrary:

Delete the MarkerLibrary.

1. GetAppVersion:

Get the latest version of the MarkerLibrary.

1. GetLECPath:

Get the LECPath, use the switch\_case suit to find out the type of the destination.

1. GetFontRecordProps:

Call the FontRecordToByte from the FontManager in the JobManager, which will packages the object into bytes and copied bytes to the destination location. Here consider the InstalledFontRecord class as an object.

1. ClearDetectedDeviceList:

First wait for the synchronization of the MutexDeviceList, and then use the iterator to delete each element in the list.

Config Record functions:

(Usually we only consider the Laser and Lens configure record.)

1. GetConfigRecordCount:

Get the number of the configure file in the corresponding list. Either LaserList or LensList.

1. NewConfigRecord:

Creates a new config object with default settings and copies it to an XML file in the “library”.

1. CloneConfigRecord:

Creates a clone of an existing config object and copies it to an XML file in the “library”.

1. SaveConfigRecordToFile:

Get the corresponding record by reference and convert it to XML.

1. DeleteConfigRecord:

Deletes an existing config object from the list and deletes the associated XML file from the library.

1. SetConfigRecordProps:

Call the ConfigRecordFromBytes in the ConfigManager, de-serializes the bytes sent from managed code into the config object.

1. GetConfigRecordProps:

Call the ConfigRecordToBytes function and serializes the config object into bytes to be sent to managed code.

1. SetActiveConfigRecord:

Call the ConnectivityManager’s GetLECDeviceConnection function, first find the device by reference number, try and get the connection, the call the ConfigManager’s SetActiveConfigRecord function and pass the device connection as a parameter. In this function, it will first get the corresponding configure by reference as the base configure and then stream it to the hardware. The StreamToHardware function in the LensConfig.cpp file documents that.

1. GetActiveConfigRecordRef:

Job Funtions:

1. NewJob:

Call the NewJob function in the JobManager, it will first detect if a job with the same name is already loaded, if so , we delete the existing job first and then create a new job by the given job name; otherwise we create the new job directly. Then add the new created job to the JobList and save the active job by “FActiveJob = AJobIndex”

1. LoadJobFromFile:

First build the file path, then detect if a job with the same name is already loaded, then create a new job from file, which first create a new job and call the LoadFromFile function to load the file.

1. SaveJobToFile:

This will call the SaveJobToFile function in the LECJobManager, first it will call the SetActiveJobByRef, set the current job to be the active one. Then call the SaveToFile function of the active job and save it to the destination.

1. CloseJob:

It will call the DeleteJob function in the MarkerLibrary and in this function it will first check if the FIFO is currently loading, because it is not allowed to delete a file while it is being loading. If not, it will call the DeleteJob function in the JobManager to delete the job by reference.

1. SetJobProps:

It will first set the job with the given reference as the active job, and then get the reference of the current active job, and then set the properties of the job from the given bytes area, that means extract the properties from the bytes.

1. GetJobProps:
2. AbortJob:

In the JobManager, it will first latch the abort state by calling GetProcessLoopState, and set it to be Idle; the it will call the AbortMotion function in the MotionManager which will go through the entire axis in the motion manager and abort them. And call the BaseDeviceConnection->BaseHAL->Abort function

1. Snapshot:
2. Undo:
3. Redo:

Object Functions:

(When we want to modify some information of the property or object or job, we follow the steps:

First set the active job to be the one with the given reference number, and get the pointer to the active job; then if we want to modify the information within the job, we do that directly, otherwise we search through the objlist to find the particular object and do the modification.)

1. GetObjectCount:

First set the active job to be the one with the reference parameter, then as before get the pointer to the active job and then call the GetObjectCount function the get the object within the job.

1. AddLECObject:

Set and find the job with the reference number as the active job and then call the AddLECObject function within it. Within that function, it will first check what is the object type, and then it will creates a new object and append it to the object list of the current job. If there is a byte array representing the vector list, then call the VectorsFromBytes function to de-serialize it into the object. Then add this object to the ObjList of the job, then we call the initialization function of the job, and then pass back the object ref and the object index to the caller.

1. DeleteLECObject:

Call the RemoveLECBaseByRef function in the LECJob, which search for the name of that object and then erase that.

1. TransformLECObject:

Call the TransformFromBytes function of the object and within this function, it will first copy the rotation bytes to the corresponding data area, and convert the matrix bytes, and call the TransformAllLists function, and then invalidate the main instruction list.

1. SetLECObjectProps:

It will first de-serialize the object by calling the FromBytes function which will copy the corresponding materials to the properties in the job.

1. GetLECObjectProps:
2. SetLECObjectVectors:

Call the VectorsFromBytes function in the object, save the list size, delete the existing lists if they exist, create new storage and copy the data. At the end, invalidate the MainInstructionList.

1. GetLECObjectVectors:
2. ExecuteObjects:

Why should we clear the vObjList first and then push the AObjList back, since we have already got the active job.

First set the referenced job to be the active one and then grab a reference to the device connection, then call the JobManager’s ExecuteObjects function, which will first grab the active job, the connection client, and the repeat flag. Then clear the ObjList and refill it with the objects in the AObjects. Then signal the event to wake up the thread and execute the job.

* The parameters \* ADeviceID is to represent which device should the objects be executed on;
* AJobRefs represents which Job does these objects belong to.
* \* AObjRefs given the address or index within the Job.

Since the vObjList in the LECJobManager is the list holding the LECBaseRT objects that to be executed, thus we should first clear the list and then put the targeted job’s targeted objects to this list, and then signal the handle.

Buffer Management Functions:

1. InitializeBufferManager:

Call the Init function in the RingBuffer, which initializes the NumSegments and SizeOfSegment, and calculate the BufferSize by multiplying these two arugments. Then allocate these buffers and create the pointer to the buffer. And for each segment in the array, call the init function, which will set the size of the segment to be zero and the offset.

1. GetLECDeviceIDList:

For each device in the list, create a new char which can store the device ID and copy the device ID to this place.

1. GetLECDeviceProperties:

Get the reference of the device and call the GetProperties function of this device.

1. DisconnectLECDevice:

Get the reference of the device and first synchronize the ConnectedDeviceMutex, then grab the current object count. Once get that, change the state of the detected device to available and then call the disconnect function to disconnect it. Then delete that device, erase that device from the LECDeviceConnectionList, and then release the mutex.

1. LoadLensData:
2. LoadLaserData:
3. GetDigitalIO:

Get a device connection and grab the data, that means call the ReadDIORegister function and read in the data.

1. GetAnalogIO:

Get a device connection and grab the data, that means call the ReadDIORegister function and read in the data.

1. ReadRegister:
2. WriteRegister:

List Management functions:

1. GetListSize:??(is this the actual size of a single segment?)
2. ExecuteList:

Call the ExecuteSegment function which will execute the segment with the corresponding index number.

1. LoopList:

Call the LoopSegment function which will first get a device connection and then call the function of the device to send data.

1. AbortList:

Get a device connection and then call the device’s HAL’s abort function

List Commends:

(51-98 they are the same as that in 51, they all first call the RingBuffer’s fpgacontrol::corresponding function, and this function just copy some information into the segment. The only difference is that the information they want to put into the segment have different opcodes thus may have different sizes.  
 And different functions have different parameters thus some might need more space to store extra data.)

1. Application\_Event:

Call the application \_Event in the RingBuffer, then call the fpgacontrol::Application\_Event to store the information and then add the data to the segment.

1. Begin\_Job\_Event:
2. Enable\_MOTF:
3. Enable\_Wobble:
4. End\_Job\_Event:
5. Job\_Marker:
6. Jump\_Abs:
7. Laser\_On:
8. Laser\_Signal\_Off:
9. Laser\_Signal\_On:
10. Long\_Delay:
11. Mark\_Abs:
12. MOTF\_Reset\_Jump:
13. Set\_Active\_Correction\_Table:
14. Set\_Correction\_Table:
15. Set\_Galvo\_DAC\_Config:
16. Set\_Jump\_Speed:
17. Set\_Laser\_Enable\_Delay:
18. Set\_Laser\_Enable\_Timeout:
19. Set\_Laser\_FPK:
20. Set\_Laser\_Mod\_Delay:
21. Set\_Laser\_Mode\_Config:
22. Set\_Laser\_Off\_Delay:
23. Set\_Laser\_On\_Delay:
24. Set\_Laser\_Pipeline\_Delay:
25. Set\_Laser\_Power:
26. Set\_Laser\_Power\_Change\_Delay:
27. Set\_Laser\_Pulse:
28. Set\_Laser\_Standby:
29. Set\_Laser\_Timing:
30. Set\_Mark\_Speed:
31. Set\_Microvector\_Delay:
32. Set\_MOTF\_Direction:
33. Set\_MOTF\_Cal\_Factor:
34. Set\_MOTF\_Mode:
35. Set\_PM\_FieldOffset:
36. Set\_PM\_FieldOrientation:
37. Set\_PM\_LaserPowerPct:
38. Set\_PM\_MarkSpeedPct:
39. Set\_PM\_PulsePeriodPct:
40. Set\_PM\_PulseWidthPct:
41. Set\_Servo\_Config:
42. Set\_Wobble:
43. Wait\_For\_IO:
44. Wait\_For\_MOTF\_Count:
45. Write\_Analog:
46. Write\_Digital:
47. Z\_Abs:

Differences between the ExecuteSegment and LoopSegment:

(Here pDevice is a device connection)

1. In the ExecuteSegment function, it calls the RingBufferManager’s ExecuteSegment function:

pRingBuffer->ExecuteSegment(pDevice, nIndex);

and it calls the SendData function of the device:

pDevice->pHAL->SendData(NULL,(unsigned char\*)GetSegmentAddr(nIndex),pSegments[nIndex].GetSize(),true,false);

And in the SendData function,

SendData(wchar\_t\* ASender, unsigned char\* AByteArray, int AListSize, bool AForce, bool ALoop)

Since the ALoop is set to be false, it will call the WriteData function of the device:

WriteData(AByteArray, AListSize)

And then in the WriteData function, it will first package the pure data with a Header by calling the PackageStreamData function:

PackageStreamData(header, (unsigned char\*)AByteArray, AListSize, &newbytearray, &newlistsize);

After they store all the data into the newbytearray, we loop to send the data.

After returned to the SendData function, call the SetEvent function to signal the hAvailable attribute.

1. In the LoopSegment function, it calls the functions of the device directly:

pDevice->pHAL->SendData(NULL, (unsigned char\*)segment, 0, true, true);

Since the ALoop is set to be true, we first convert the AByteArray into the LoopingSegment, it will create the pointer to the all the segments to loop, which is also an attribute in the BaseHAL, then we create the thread to execute the looping segment:

CreateThread(NULL, 0, StartExecutionThread, this, 0, NULL)

And then call the SetEvent function to signal the hRunloop attribute.

Both the hAvailable and hRunloop are attributes in the BaseHAL class.

但是为什么execute要call ring buffer manager的函数，然而loop就直接call device呢？而且传的参数不同size那里