README

Friday, October 27, 2017

# Source Code Documentation

This document describes some of the in-house developed source code used in deploying the UR MTConnect agent. The code was developed and compiled using Microsoft Visual C++ 2010. Since MSVC 2010 predates hte C11 standard, many useful programming primitives (bind, mutex, thread) relied on the boost 1.54 library instead of the std library in C11. The rationale for keeping MSVC 2010 is the support of the MSI deployment project, so that after the UR Agent was coded and tested, it could be packaged and deployed as a MSI install script.

Some caveats are in order. The release build did not operate correctly, although the Debug build operated correctly. In order to fix this, ALL optimization flags were turned off in the Release build and this fixed the problems. These optimizations flags appear in the C++/C Optimization properties and include:

* /GL Turnn off whgole program optimization
* /O2 /Oy- which optimized the code for speed, size etc were changed to none.

## Struct Decoder

The struct decoder performs conversions between network byte arrays and C++ structs. This intended for use in handling binary data from network connections. First, all the structure variables are declared. Then, preprocessor macros are used to define the size and order of the byte stream associated with this C++ struct. It assumes the struct is packed with no padding between variable elements. g each entry so they can easily be read/write/print including byte swapping if required. Implicit in this encoding process is the determination of the byte order of the host machine versus the prevailing network byte order (Big Endian), to determine whether byte swapping is necessary. If the byte order is not the same as network byte order, bytes are swapped to accommodate network byte order.

The struct\_decoder struct is useful for reading/writing network byte streams. It uses the preprocessor definition to define a property table describing. The struct\_decoder relies a printf specifier to format the variable for output, which conforms to the following table:

| **Format** | **C Type** | **Standard size**  **bytes** | **Notes** |
| --- | --- | --- | --- |
| %c | **Signed char** | 1 |  |
| %c | **unsigned char** | 1 |  |
| %uc | **unsigned char** | 1 |  |
| %c | **bool** | 1 |  |
| %u | **short** | 2 |  |
| %hu | **unsigned short** | 2 |  |
| %d | **int** | 4 |  |
| %ud | **unsigned int** | 4 |  |
| %ld | **long** | 4 |  |
| %uld | **unsigned long** | 4 |  |
| %lld | **long long** | 8 |  |
| %ulld | **unsigned long long** | 8 |  |
| %f | **float** | 4 |  |
| %lf | **double** | 8 |  |
| %Lf | **long double** | 16 |  |
| %s | **char[]** |  |  |

Array serial properties use these printf to format each element in the array.

The "Curiously Recurring Template" pattern is used to assist in decoding by allowing access to the outermost serial property map. Thus, each structure that uses the serialization decoding mechanism, inherits from "struct\_decoder" template class, but provides itself as the template type. This is useful if the structure inherits from a structure that itself has a serial property map.

struct ur\_cartesian\_info : public struct\_decoder<ur\_cartesian\_info>

The "Curiously Recurring Template" is useful when you need to access the outer class variables or methods. This is used to get the outer serial property map defined in the class, as shown in the struct\_decoder base class:

template <class T>   
struct struct\_decoder

{

. . .

PropertyMap \*props = static\_cast<T \*>(this)->GetPropertyMap();

There are a collection of C++ macros that handle the building of the property map that are used by the "Curiously Recurring Template" struct\_decoder. These include:

* BEGIN\_SERIAL\_PROP\_MAP (X) marks the beginning of the classes' serial property map. It creates a msgname class to return ASCII string containing the message name (denoted by the preprocessor X). It also declares a the beginning of a static method GetPropertyMap(), which gives a reference pointer to the embedded static table of serialization variables.
* PROP\_SERIAL\_ENTRY (mName, pformat) defines a preprocessor macro that inserts an entry into the serial property map table. It expects an existing class member variable name, and uses the mName to save the name, sizeof, and offset of the variable from beginning of the class definition (including any inherited classes definitions). Pformat defines a printf like formatting string for use in the print struct\_decoder method.
* PROP\_SERIAL\_ENTRY\_VERSION(mName, pformat,min,max) defines a preprocessor macro that inserts an entry into the serial property map table, much like PROP\_SERIAL\_ENTRY. However, this time it also enter a min and max version number in floating point (e.g., 3.4) into the serial property table. VMAX and VMIN are macros to indicate the limits of the versioning in either direction.
* PROP\_SERIAL\_ENTRY\_VARARRAY(mName, mSize, pformat, var) defines a preprocessor macro that inserts an array entry into the serial property map table. mName and pformat are used as int PROP\_SERIAL\_ENTRY. mSize defines the size of each element in the array, will the var entry defining a pointer to the variable defining the actual size of the array. For example, below DHa is the name of the variable, and element size is the sizeof(double), the format is "%lf" for a double, and the pointer to the defining array size is given by &numjoints.

PROP\_SERIAL\_ENTRY\_VARARRAY(DHa, sizeof( double ), "%lf", &numjoints)

Note, the max size of the array is given by the sizeof(DHa) so it must be allocated as a declared array size, not as a std::vector. Using this information, we can insure that the size given by numjoints does not exceed the maximum declared array size.

double DHa[8];

* END\_SERIAL\_PROP\_MAP() marks the end of the serial property map.
* PROP\_SERIAL\_ENTRY\_BASE(mName) allows the current structure to reference an inherited structure with a serial property map of its variables. It is no used in the UR Agent implementation.

For example, the class ur\_cartesian\_info illustrates the use of the inheritance, range of applicability using minimum and maximum verion numbers, use of pragma to insure variables are packed,

struct ur\_cartesian\_info : public struct\_decoder<ur\_cartesian\_info>

{

float mVersion;

#pragma pack(push, 1)

double X;

double Y;

double Z;

double Rx;

double Ry;

double Rz;

double TCPOffsetX;

double TCPOffsetY;

double TCPOffsetZ;

double TCPOffsetRx;

double TCPOffsetRy;

double TCPOffsetRz;

#pragma pack(pop)

BEGIN\_SERIAL\_PROP\_MAP(ur\_cartesian\_info)

PROP\_SERIAL\_ENTRY(X, "%lf")

PROP\_SERIAL\_ENTRY(Y, "%lf")

PROP\_SERIAL\_ENTRY(Z, "%lf")

PROP\_SERIAL\_ENTRY(Rx, "%lf")

PROP\_SERIAL\_ENTRY(Ry, "%lf")

PROP\_SERIAL\_ENTRY(Rz, "%lf")

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetX, "%lf",3.1,VMAX)

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetY, "%lf",3.1,VMAX)

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetZ, "%lf",3.1,VMAX)

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetRx, "%lf",3.1,VMAX)

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetRy, "%lf",3.1,VMAX)

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetRz, "%lf",3.1,VMAX)

END\_SERIAL\_PROP\_MAP( )

} ;

The defines the structure and uses the

struct ur\_cartesian\_info : public struct\_decoder<ur\_cartesian\_info>

{

The mVersion variable must be inserted into every outer class so that multiple instances of the ur\_cartesian\_info struct can support multiple versions of the interface. It is manually inserted so that programmers are aware that versioning is being done, and the version must be specified. Thus a 3.0 UR robot and a 3.4 UR robot can each use the ur\_cartesian\_info definition, and by putting in either 3.0 or 3.4 different interfaces are supported during decoding.

float mVersion;

In the definition of ur\_cartesian\_info, the Microsoft C++ compiler pragma pack is used to ensure that the variables are slotted with their native size and no padding is performed. Thus, before the start of the variables to be decodes from a binary representation, the pragma is used:

#pragma pack(push, 1)

double X;

. . .

#pragma pack(pop)

The serial property map for each structure must be defined to allow the decoding (read) and the encoding (write) of raw binary message. Implicit in this transcribing process is the determination of the byte order of the host machine, to determine whether it is the same as network byte order or not. If the byte order is not the same as network byte order, bytes are swapped to accommodate network byte order.

BEGIN\_SERIAL\_PROP\_MAP(ur\_cartesian\_info)

PROP\_SERIAL\_ENTRY(X, "%lf")

. . .

END\_SERIAL\_PROP\_MAP( )

Changes due to versioning in the ur\_cartesian\_info struct occur in version 3.1 So, in the property serial map specification, the version ranges from VMIN to 3.1 to VMAX. So, the version 3.0 would not support the TCPOffsetX variable, while the version 3.1 would. Below the X variable is declared in the preprocessor macro PROP\_SERIAL\_ENTRY where the version implicitly ranges from VMIN to VMAX:

PROP\_SERIAL\_ENTRY(X, "%lf")

Below the TCPOffsetX variable is declared in the preprocessor macro PROP\_SERIAL\_ENTRY\_VERSION to be viable from version 3.1 on:

PROP\_SERIAL\_ENTRY\_VERSION(TCPOffsetX, "%lf",3.1,VMAX)

Once the structure has been setup to be decoded (and encoded), raw messages from the UR robot can be unpacked into the structure. Assuming a socket reader has been coded, we assume a variable "buf" contains a raw binary message, with a given offset (generally including or not including header) and the variable len describing the length of the message. The ur\_robot\_state::unpackRobotState accepts these parameters on the stack to decode the UR message, in the following code:

void ur\_robot\_state::unpackRobotState (uint8\_t \*buf, unsigned int offset,

uint32\_t len)

{

// package\_types

offset += 5;

while ( offset < len )

{

ur\_message\_header hdr;

hdr.read((char \*) &buf[offset]);

switch ( hdr.packageType )

{

case package\_types::CARTESIAN\_INFO:

{

\_ur\_cartesian\_info.read((char \*) &buf[offset + hdr.msglength( )]);

}

break;  
. . .

}

offset += hdr.packageSize;

}

The header offset describing the outer message type and length (of length 5 bytes) is added to the offset within the message.

// package\_types

offset += 5;

Then the header message describing the type and length of the message is decoded using the ur\_message\_header decoding "read" routine:

ur\_message\_header hdr;

hdr.read((char \*) &buf[offset]);

Using the hdr packageType can be used in a C++ switch statement to determine the type of the inner message:.

switch ( hdr.packageType )

{

We will assume that a ur\_cartesian\_info message is to be decoded. Then the package\_types::CARTESIAN\_INFO case will be matched:

case package\_types::CARTESIAN\_INFO:

{

\_ur\_cartesian\_info.read((char \*) &buf[offset + hdr.msglength( )]);

}

break;

And then the ur\_cartesian\_info.read routine will read (i.e.,decode) the buffer starting at position buf+offset with message length equal to hdr.msglength( ) that is determined dynamically each time in consideration of a potential version and resulting message length change.

# Logging

There is limited logging capability within the UR Agent. The logging facility is based on ROS Console open source code, which can be found at <https://github.com/ros/console_bridge/blob/master/include/console_bridge/console.h>. ROS provides interesting C++ Logging filtering macros (once, periodically, etc.) here: <https://github.com/ros/ros_comm/blob/4383f8fad9550836137077ed1a7120e5d3e745de/tools/rosconsole/include/ros/console.h> Although the logging facility is very robust and can be extracted out of the ROS framework with a little effort, the code base was too extensive to justify the work.

The logging is global in scope in that the all diagnostic log messages are output to the class GLogger. GLogger is defined as an instance of class CLogger. Within the instance, there are several flags which can be used to change the logging functionality. One flag is accessed by the method DebugConsole() , with 0 giving no "OuputDebugConsole" for windows, and 1 echoing the logging message to output. There is a level determined by the method DebugLevel() which determines the threshold of diagnostic activity, which is explained a bit later. The flag Timestamping( ) determines whether a timestamp will be prepended to every diagnostic message. If Timestamping( ) is true, timestamps will be appended. For example, the logStatus message with a timestamp is shown below:

2017-10-24 09:23:57.0259 Start UR\_ Agent

The CLogger variable DebugFile() determines the filename in which log messages are appended. By default the filename is debug.txt and the location of the file is in the execution application directory.

Users of applications can indeed create different log file using CLogger and DebugFile(), however, one will be limited to the logging method: logmessage (const char \*file, int line, LogLevel level, const char \*fmt, ...) to log message. Plus, the macros described below only output to GLogger, which take advantage of the logmessage method.

The logging level [-1..5] determines the amount of logging that will be done. Each numeric value corresponds to a threshold where logging will be done. 0 corresponds to Fatal, 1 corresponds to Error, 2 corresponds to Warning, 3 corresponds to Inform, and 4 corresponds to Debug.

System logging threshold omit logging with a diagnostic level above the system level. For example a system logging level of 3 will omit debug message (logging level 5)

* logAbort(fmt, ...) Something unrecoverable has happened. Application will terminate imeediately, and not gracefully.
* logFatal(fmt, ...) Something catastrophic has happened, attempt to shut the application down.
* logStatus(fmt, ...) Information that is provided regardless of logging level.
* logError(fmt, ...) Something serious (but recoverable) has gone wrong.
* logWarn(fmt, ...) Information that the user may find alarming, and may affect the output of the application, but is part of the expected working of the system.
* logInform(fmt, ...) Small amounts of information that may be useful to a user.
* logDebug(fmt, ...) Information that you never need to see if the system is working properly.

There is logging level of None but it is unclear if this is obeyed.

Below is an example of the use of the logDebug macro. It accepts a character string as format, and a variable length of arguments follow determined by the format statement. It uses stdargs to handle the variable arguments.

#include "NIST/Logger.h"

. . .

logDebug("\tAdapter %s Server Rate=%d\n", mDevice.c\_str(), mServerRate);

. . .

There is a C++ macro LOG\_ONCE(X) to limit the logging output to only once. For example, the previous code can be output once by the following code modification:

#include "NIST/Logger.h"

. . .

LOG\_ONCE (

logDebug("\tAdapter %s Server Rate=%d\n", mDevice.c\_str(), mServerRate);

)

. . .

There is also a LOG\_THROTTLE(secs, X) based on the ROS filtering methodology, this macro will limit the output to a log on a periodic basis, with the rate determined in seconds. Thus, in the example below, the diagnostic message output will be throttled to once per minute. If the time between logging outputs is greater than 60 seconds, then this log message will be output, and the logging timer will be reset.

#include "NIST/Logger.h"

. . .

LOG\_ THROTTLE (60,

logDebug("\tAdapter %s Server Rate=%d\n", mDevice.c\_str(), mServerRate);

)

. . .

In order to access the logging facility, at the beginning of the program a small preamble of code is necessary to setup and configure the logger. Below is a minimal example:

GLogger.Open(File.ExeDirectory() + "debug.txt");

GLogger.DebugLevel()=5;

GLogger.Timestamping()=true;

In this case, the output file name is found in the exe folder and named debug.txt. The default logging level is 5, although it was overridden later by a config file value. Likewise timestamping is enabled. Note, the use of method access to modify the flags (ie.e., ()). This is due to the fact that all these accessors pass a reference to the actual logger flag. Historically, this was done because it is easier to override a method than a variable in C++.

## Ini File Software

The UR Agent relies on some in-house INI configuration file management that is not typical of normal MTConnect installations, which rely on YAML. As an aside, if MTConnect had relied on JSON to provide configuration details, boost has a nice library. However, JSON is a compliant with YAML but not vice versa.

The INI file format is an informal standard for configuration based historically on the Microsoft. INI files are simple text files with a basic structure composed of sections, key/properties/tags, and values.

The basic element contained in an INI file is the key or property. Every key has a name and a value, delimited by an equals sign (=). The name appears to the left of the equals sign.

name=value

For the UR Agent configuration, keys are be grouped into named sections. The section name appears on a line by itself, in square brackets ([ and ]). All keys after the section declaration are associated with that section. There is no explicit "end of section" delimiter; sections end at the next section declaration, or the end of the file. Sections may not be nested. The term key and property and tag are used interchangeably and describe the same INI file element.

[section]

key=value

tag=value

The ini file software is managed by the class Config, and uses a simplist variant class (StringVariant) to manage the different configuration types and conversion. Note, for this implementation the ability to determine wheter string case in keys and section names is important was added. For this implementation, Config assumes section and key names are CASE INSENSITIVE. This was found to be useful as Microsoft ini file functions (e.g., WritePrivateProfileString) are case insensitive as well.

The software operation of the ini file management is done in the agent on a global level and in each adapter on a local device level. They both use the same Config.ini file for reading configuration data. The Config.ini filename is held in the variable Globals.mInifile and is tested for existence. A fatal error occurs if the file is not found, and the UR Agent logs an fatal message and terminates. If the file exists, each important key is read from a section. For the agent, this section name is "GLOBALS" and for each adapter, it is the name of the robot device (e.g., [marvin]). Then each key is read into the corresponding global or device variable.

void AgentConfigurationEx::initialize (int aArgc, const char \*aArgv[])

{

std::string cfgfile = Globals.mInifile;

if (File.Exists(cfgfile.c\_str( ))

{

mConfig.load(cfgfile);

Globals.mServerName

= mConfig.GetSymbolValue<std::string>("GLOBALS.ServiceName", Globals.mServerName);

MTConnectService::setName(Globals.mServerName);

. . .

mDevices = mConfig.GetTokens("GLOBALS.MTConnectDevice", ",");

First, the ini file must be read:

mConfig.load(cfgfile);

Once loaded, any key or section can be parsed. Each key/value pair is read, and then the method GetSymbolValue is a template method which uses the template type in order to convert the underlying Config variant type into the given type (in this example the template typename is std::string). Importantly for MTConnect, one can name the UR Agent service and then store this name into the MTConnect Institute core agent:

Globals.mServerName

= mConfig.GetSymbolValue<std::string>("GLOBALS.ServiceName", Globals.mServerName);

MTConnectService::setName(Globals.mServerName);

The Config file also has the method to parse a token list (e.g., comma separated values) and return a standard vector of string. Thus GetTokens reads the section "[GLOBALS]" and the key "MTConnectDevice" first as a key/value pair, and then parses the value using the "," comma separator. This mDevices variable ultimately contains the list of devices for which an adapter must be created.

mDevices = mConfig.GetTokens("GLOBALS.MTConnectDevice", ",");

The Config class is simplistic way of dealing with configuration data, but only requires a modest amount of code, and most every programmer is familiar with ini files.

## Reset at Midnight

MTConnect Agents are expected to operate continually 24/7 without fault. Because some underlying software technology may leak or consume memory with garbage collection so that performance degrades, it became apparent that have the capability to reset the agent at midnight requiring a few seconds of down time, every night, cleared the system of the potential software downsides.

The CResetAtMidnightThread class was coded to handle the reset at midnight behavior. It uses the Microsoft CWorkerThread template from the ATL, which is a class that creates a worker thread, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled. The condition that the CWorkerThread waits on is a timer which expires at midnight. Upon expiring, the CWorkerThread runs and contains code to spawn a Process which reset the service (as given by the service name in the configuration file).

The heart of the WorkerThread declaration is given below:

CWorkerThread<> \_resetthread;

struct CResetThread : public IWorkerThreadClient

{

HRESULT Execute(DWORD\_PTR dwParam, HANDLE hObject);

HRESULT CloseHandle(HANDLE){ ::CloseHandle(\_hTimer); return S\_OK; }

HANDLE \_hTimer;

} \_ResetThread;

The variable \_resetthread is declared as a CWorkerThread with no template parameter to define further thread characteristics:

CWorkerThread<> \_resetthread;

Microsoft uses a leading I on a class definition to indicate that he class defines an interface, which is in fact similar to an abstract base class in C++. Then a CResetThread is declared which implements the required IWorkerThreadClient abstract methods defined in the interface. These include Execute and CloseHandle. CloseHandle closes the thread handle upon exit.

The Execute function does the actual service restart. In order to run the reset thread, it must be initialzed and a timer is added to the implementation that waits until midnight before running. Below is the complete code to start the \_resetthread. We use the flag bResetAtMidnight to insure that the reset at midnight operation has been enabled.

inline bool CResetAtMidnightThread<T>::Start()

{

if(\_bResetAtMidnight)

{

COleDateTime now = COleDateTime::GetCurrentTime();

COleDateTime date2 = COleDateTime(now.GetYear(), now.GetMonth(), now.GetDay(), 0, 0, 0) + COleDateTimeSpan(1, 0, 0, 1);

COleDateTimeSpan tilmidnight = date2-now;

\_resetthread.Initialize();

\_resetthread.AddTimer(

(long) tilmidnight.GetTotalSeconds() \* 1000,

&\_ResetThread,

(DWORD\_PTR) this,

&\_ResetThread.\_hTimer // stored newly created timer handle

) ;

return true;

}

return false;

}

The start code must compute the difference in time between now and midnight in order for the timer to fire correctly. To do this the classes COleDateTime and COleDateTimeSpan were used. COleDateTime has a static variable GetCurrentTime() to retrieve the current time

COleDateTime now = COleDateTime::GetCurrentTime();

Then the next day at one minute past midnight is computed. It takes the now timestamp, zeros out the hours, minutes and seconds, and adds one day and one mintue to the date-time. If we subtract now from the future midnight into a COleDateTimeSpan, we get the amount of time until then.

COleDateTime date2 = COleDateTime(now.GetYear(), now.GetMonth(), now.GetDay(), 0, 0, 0) + COleDateTimeSpan(1, 0, 0, 1);

COleDateTimeSpan tilmidnight = date2-now;

After initializing the CWorkerThread in the suspended state (by default). We use this COleDateTimeSpan time tilmidnight as a timer (using addTimer method) to indicate when the thread will execute, and we pass in this CWorkerThread class definition and the this pointer to the instance of the CWorkerThread.

\_resetthread.Initialize();

\_resetthread.AddTimer(

(long) tilmidnight.GetTotalSeconds() \* 1000,

&\_ResetThread,

(DWORD\_PTR) this,

&\_ResetThread.\_hTimer // stored newly created timer handle

) ;

When the timer expires, the Execute method of the CWorkerThread will be executed. The "this" pointer is passed in as the calling parameter dwParam. We first save the "this" pointer, and then cancel the existing timer using its handle given in hObject. Then we use boilerplate windows process code to create a process that will cause the Service Control Manager (SCM) to reset the named service.

template<typename T>

inline HRESULT CResetAtMidnightThread<T>::CResetThread::Execute(DWORD\_PTR dwParam, HANDLE hObject)

{

T \* service = (T\*) dwParam;

CancelWaitableTimer(hObject);

try {

PROCESS\_INFORMATION pi;

ZeroMemory( &pi, sizeof(pi) );

STARTUPINFO si;

ZeroMemory( &si, sizeof(si) );

si.cb = sizeof(si);

si.dwFlags = STARTF\_USESHOWWINDOW;

si.wShowWindow = SW\_HIDE; // set the window display to HIDE

// SCM reset command of this service

std::string cmd = StdStringFormat("cmd /c net stop \"%s\" & net start \"%s\"", service->name(), service->name()); // Command line

if(!::CreateProcess( NULL, // No module name (use command line)

const\_cast<char \*>(cmd.c\_str()),

NULL, // Process handle not inheritable

NULL, // Thread handle not inheritable

FALSE, // Set handle inheritance to FALSE

0, // No creation flags

NULL, // Use parent's environment block

NULL, // Use parent's starting directory

&si, // Pointer to STARTUPINFO structure

&pi )) // Pointer to PROCESS\_INFORMATION structure

AtlTrace("CreateProcess FAIL ") ;

::Sleep(5000); // make sure process has spawned before killing thread

}

catch(...)

{

service->Fatal("Exception - ResetAtMidnightThread(void \*oObject");

}

return S\_OK;

}

A string is built to execute in the process which starts with the cmd or DOS shell that call net stop of the service, and then uses net start of the same service.

std::string cmd = StdStringFormat("cmd /c net stop \"%s\" & net start \"%s\"", service->name(), service->name());

This brief stop and start takes only seconds, however, seems to rid the system of software memory leaks, insistent memory consumption, as well as general software malaise at no real cost to the agent operation. This coding strategy can be used on Windows for exe application as well. Of note, a similar construct was not found in Linux.

## Versioning

There is a class CProductVersion to handle version identification for archival purposes. It is rather primitive. CProductVersion is class used by UR Agent that generates a string based on all constituent components used in this agent. The version information is useful for archiving what components were used to generated the code, if the source code is not available, or compatibilities issues arise. The output of the version information is displayed as an extension in the MTConnect core agent. You would access the version information with the following sample URL (assuming you are on the agent machine and the agent port is listening to port 5000): <http://127.0.0.1:5000/version>. The following non-XML but HTML table would be returned:

|  |
| --- |
| Version |
| MTConnect Core Agent Version 1.2.0.0 - built on Tue Oct 24 08:30:22 2017 |
| UR\_ Agent Extensions 64 bit Platform Version 1.3.0.60 - built on Tue Oct 24 08:30:22 2017 |
| UR\_ Agent MSI Install Version 1.0.3 |
| Windows Version : Microsoft Windows 7 Enterprise Edition Service Pack 1 (build 7601), 64-bit |
| Microsoft Visual C++ Version Visual Studio 2010 - MSVC++ 10.0 |
| XML Lib version 2.7.7 |
| Boost libraries 1.54.0 |
| UR\_ Devices = Woodsy |

## Use of NIST Information

This document is provided as a public service by the National Institute of Standards and Technology (NIST). With the exception of material marked as copyrighted, information presented on these pages is considered public information and may be distributed or copied. Use of appropriate byline/photo/image credits is requested.

## Software Disclaimer

NIST-developed software is provided by NIST as a public service. You may use, copy and distribute copies of the software in any medium, provided that you keep intact this entire notice. You may improve, modify and create derivative works of the software or any portion of the software, and you may copy and distribute such modifications or works. Modified works should carry a notice stating that you changed the software and should note the date and nature of any such change. Please explicitly acknowledge the National Institute of Standards and Technology as the source of the software.

NIST-developed software is expressly provided “AS IS.” NIST MAKES NO WARRANTY OF ANY KIND, EXPRESS, IMPLIED, IN FACT OR ARISING BY OPERATION OF LAW, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT AND DATA ACCURACY. NIST NEITHER REPRESENTS NOR WARRANTS THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT ANY DEFECTS WILL BE CORRECTED. NIST DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE SOFTWARE OR THE RESULTS THEREOF, INCLUDING BUT NOT LIMITED TO THE CORRECTNESS, ACCURACY, RELIABILITY, OR USEFULNESS OF THE SOFTWARE.

You are solely responsible for determining the appropriateness of using and distributing the software and you assume all risks associated with its use, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and the unavailability or interruption of operation. This software is not intended to be used in any situation where a failure could cause risk of injury or damage to property. The software developed by NIST employees is not subject to copyright protection within the United States.