Socket Stream Replay Documentation

Wednesday, November 08, 2017

This document describes the replay of timestamped socket capture on a given socket using a capture filename. This is heopful in testing status socket streams. Status socket streams will be defined as TCP/IP streams that cyclically emit a stream as one or more packets of status information. This package can play back a recorded status stream so the programmer does not need the actual device to test against. For example, a real status device may emit a stream on device 30002 with an IP associated with the controller. After recording the steram, it can be played back on the current machine at a given port. Typically one would choose the iIP/port pair 127.0.0.1 and 3002 to match the actual device port, but it is not mandatory.

The capture file format is timestamp|hex transliteration of each character …. \n i.e.,

Timestamp | xx, xx, . . . xx, \n

Thus, each packet is first timestamped and then translated into a series of hex characters matching its binary equivalent character in the stream, and a line feed is appended at the end of the packet. The socket capture is thus a timestamp series of packet captures especially useful for raw binary communication. Below illustrates two timestamped lines of a socket capture.

2017-11-03 09:00:09.0471 |00,00,00,90,00,00,00,0F,00,00,00,01,00,00,00,00,00,00,00,00,00,00,00,02,00,00,00,00,3F,4B,11,AF,3E,C1,FE,4D,00,00,00,00,BF,F7,DA,1F,BA,56,FF,35,BF,52,0B,20,BF,65,F7,BA,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,

2017-11-03 09:00:09.0482 |00,00,00,28,00,00,00,0D,00,00,00,01,00,00,00,00,00,00,00,00,00,00,00,01,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,02,00,00,00,00,

Replay means the timestamps are interpreted as the time difference between transmitting the packets across the given socket, assuming there are listeners.

Table of Contents

[Configuring the Source Code for Compilation 2](#_Toc497908007)

[Logging 3](#_Toc497908008)

[Ini File Software 4](#_Toc497908009)

[DISCLAIMER 6](#_Toc497908010)

[Use of NIST Information 6](#_Toc497908011)

[Software Disclaimer 6](#_Toc497908012)

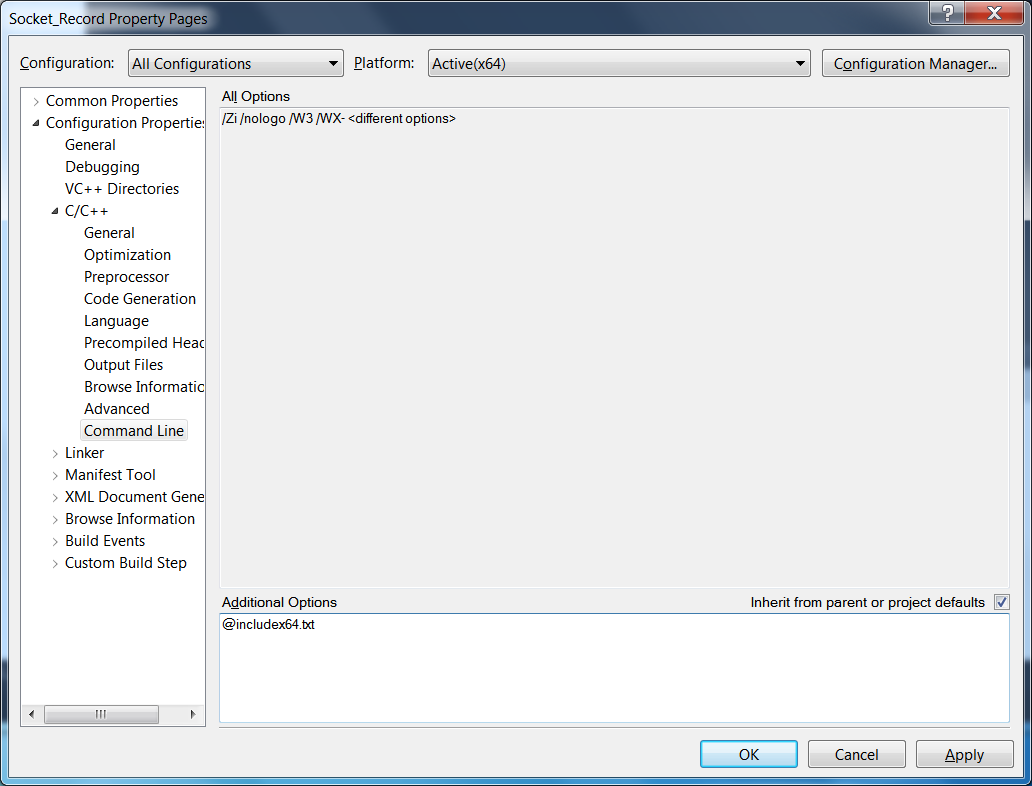
# Configuring the Source Code for Compilation

Cofniguring the source code for compilation and linking requires a two part manual tweaking operation. This is due to the fact that the include and link libraries paths are hard coded. This is not elegant, but is much easier to ascertain when things are wrong.

First, you must set up the include file so that all the include files can be found properly and second the link module must be modified so that the boost libraries can be found.

Add the preprocess definition: \_WINDOWS

## C++ Header Configuration



## Link Configuration

To customize the source code there is a linking module in the project, named LinkingModule.cpp, that uses the "#pragma comment(lib. . ." MSVC construct to declare what libraries to link against. The software uses boost and thus needs to know where the boost libraries are located. Thus, the macros BOOSTLIBPATH32 and BOOSTLIBPATH64 were used to assist in defining the folder location of the boost libraries. Below, the macros are defined for the 64-bit machine:

if 0

#define BOOSTLIBPATH32(X) \

"D:\\Program Files\\NIST\\src\\boost\_1\_54\_0\\stageX32\\lib\\"##X

#else

#define BOOSTLIBPATH64(X) \

"C:\\Program Files\\NIST\\src\\boost\_1\_54\_0\\vc10\\stagex64\\lib\\"##X

#define BOOSTLIBPATH32(X) \

"C:\\Program Files\\NIST\\src\\boost\_1\_54\_0\\vc10\\stagex32\\lib\\"##X

#endif

One could of course enhance the macro differentiation. Then, depending on the build configuration (debug versus release, 64 bit versus 32 bit) different boost libraries are used. Below the preprocessor definition #if is used to differentiate the different architectures, which assume the preprocessor definitions WIN64, \_DEBUG, WIN32 are defined appropriately in the processor configuration settings in MSVC.

#if defined( WIN64 ) && defined( \_DEBUG )

#pragma message("DEBUG x64")

#pragma comment(lib, BOOSTLIBPATH64("libboost\_system-vc100-mt-sgd-1\_54.lib"))

#pragma comment(lib, BOOSTLIBPATH64("libboost\_thread-vc100-mt-sgd-1\_54.lib"))

#elif !defined( \_DEBUG ) && defined( WIN64 )

#pragma message("RELEASE x64")

#pragma comment(lib, BOOSTLIBPATH64("libboost\_thread-vc100-mt-s-1\_54.lib"))

#pragma comment(lib, BOOSTLIBPATH64("libboost\_system-vc100-mt-s-1\_54.lib"))

#elif defined( \_DEBUG ) && defined( WIN32 )

#pragma message("DEBUG x32")

#pragma comment(lib, BOOSTLIBPATH32("libboost\_system-vc100-mt-sgd-1\_54.lib"))

#pragma comment(lib, BOOSTLIBPATH32("libboost\_thread-vc100-mt-sgd-1\_54.lib"))

#elif !defined( \_DEBUG ) && defined( WIN32 )

#pragma message("RELEASE x32")

#pragma comment(lib, BOOSTLIBPATH32("libboost\_thread-vc100-mt-s-1\_54.lib"))

#pragma comment(lib, BOOSTLIBPATH32("libboost\_system-vc100-mt-s-1\_54.lib"))

#endif

# 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 strategies, 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.

# DISCLAIMER

## 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.