MTConnect Installation with NSIS

John Michaloski  
Thursday, December 04, 2014 11:09:16 AM

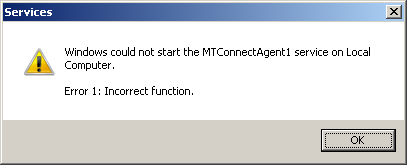
# MTConnect Install Warnings

The following is list of potential problems associated with the MTConnect agent install.

* Overwriting of existing files – although NSIS has the capability to overwrite existing files by   
  SetOverwrite on

This has not found to be reliable – but it could be pilot error in misunderstanding how to achieve the overwriting. Users are encouraged to run uninstall.exe in the install folder and then run a clear install to make sure the install works properly. The uninstall could be run in the install script, but is not at this time.

* You MUST INSTALL the MTConnect Agent with “agent.exe install” or the Windows registry is not properly setup and then subsequent starting of the service will fail. You will get this ominous pop up message box:



One might be tempted to use the service registry function (sc.exe) to install the agent, with the code below (BUT DON’T).

sc.exe create MTConnectAgent start= auto binpath= "$INSTDIR\Agent.exe”

Thus, the “agent install”l functionality was inserted after configuration has completed, in the last function (.onGUIEnd).

Function .onGUIEnd

;ExecWait 'sc.exe create $ServiceName start= auto binpath= "$INSTDIR\Agent.exe"'

ExecWait '"$INSTDIR\Agent.exe" install '

${If} $AgentserviceStart <> 0

ExecWait 'sc.exe start $ServiceName' $0

${EndIf}

...

FunctionEnd

This is the code in the agent install that must be executed for the agent start to work:

void MTConnectService::install()

{

...

#if 0

HKEY software;

LONG res = RegOpenKey(HKEY\_LOCAL\_MACHINE, "SOFTWARE", &software);

if (res != ERROR\_SUCCESS)

{

sLogger << dlib::LERROR << "Could not open software key (" << res << ")";

return;

}

HKEY mtc;

res = RegOpenKey(software, "MTConnect", &mtc);

if (res != ERROR\_SUCCESS)

{

res = RegCreateKey(software, "MTConnect", &mtc);

RegCloseKey(software);

if (res != ERROR\_SUCCESS)

{

sLogger << dlib::LERROR << "Could not create MTConnect (" << res << ")";

return;

}

}

RegCloseKey(software);

// Create Service Key

HKEY agent;

res = RegOpenKey(mtc, mName.c\_str(), &agent);

if (res != ERROR\_SUCCESS)

{

res = RegCreateKey(mtc, mName.c\_str(), &agent);

if (res != ERROR\_SUCCESS)

{

RegCloseKey(mtc);

sLogger << dlib::LERROR << "Could not create " << mName << " (" << res << ")";

return;

}

}

RegCloseKey(mtc);

// Fully qualify the configuration file name.

if (mConfigFile[0] != '/' && mConfigFile[0] != '\\' && mConfigFile[1] != ':')

{

// Relative file name

//char path[MAX\_PATH];

//GetCurrentDirectory(MAX\_PATH, path);// not necessarily in current directory, alway in exe path

mConfigFile = ((std::string) path) + "\\" + mConfigFile;

}

RegSetValueEx(agent, "ConfigurationFile", 0, REG\_SZ, (const BYTE\*) mConfigFile.c\_str(),

mConfigFile.size() + 1);

RegCloseKey(agent);

#endif

The code in the agent start service that fails is below highlighted in italics:

VOID SvcInit( DWORD dwArgc, LPTSTR \*lpszArgv)

{

char path[MAX\_PATH];

if( !GetModuleFileName(NULL, path, MAX\_PATH ) )

{

Trace("SvcInit - Cannot get path of executable\n" );

sLogger << dlib::LERROR << "Cannot get path of executable (" << GetLastError() << ")";

return;

}

#if 0

***// Get the real arguments from the registry***

***char key[1024];***

***snprintf(key, 1023, "SOFTWARE\\MTConnect\\%s", gService->name().c\_str());***

***HKEY agent;***

***LONG res = RegOpenKeyEx(HKEY\_LOCAL\_MACHINE, key, 0, KEY\_READ, &agent);***

***if (res != ERROR\_SUCCESS)***

***gService->install();***

***res = RegOpenKeyEx(HKEY\_LOCAL\_MACHINE, key, 0, KEY\_READ, &agent);***

***if (res != ERROR\_SUCCESS)***

***{***

***OutputDebugString("RegOpenKey: Could not open MTConnect Agent Key\n");***

***SvcReportEvent("RegOpenKey: Could not open MTConnect Agent Key");***

***ReportSvcStatus( SERVICE\_STOPPED, 1, 0 );***

***return;***

***}***

***const char \*argp[2];***

***BYTE configFile[2048];***

***DWORD len = sizeof(configFile) - 1, type;***

***res = RegQueryValueEx(agent, "ConfigurationFile", 0, &type, (BYTE\*) configFile, &len);***

***RegCloseKey(agent);***

***if (res != ERROR\_SUCCESS)***

***{***

***OutputDebugString("RegOpenKey: Could not open ConfigurationFile\n");***

***SvcReportEvent("RegOpenKey: Could not open ConfigurationFile");***

***ReportSvcStatus( SERVICE\_STOPPED, 1, 0 );***

***return;***

***}***

***argp[0] = (char\*) configFile;***

***argp[1] = 0;***

***gService->initialize(1, argp);***

#endif

const char \*argp[2];

argp[0] = (std::string(path)+"\\Agent.cfg").c\_str();

argp[1] = 0;

gService->initialize(1, argp);

// Report running status when initialization is complete.

ReportSvcStatus( SERVICE\_RUNNING, NO\_ERROR, 0 );

gService->start();

ReportSvcStatus( SERVICE\_STOPPED, NO\_ERROR, 0 );

}

# MTConnect Install Requirements

MTConnect is an integration standard to solve the “Island of Automation” problem in the discrete manufacturing industry. MTConnect is an open, royalty-free standard that uses prevalent COTS technology - XML and HTTP. MTConnect was developed to facilitate the exchange of data on the manufacturing floor and was initially sponsored by the Association for Manufacturing Technology (AMT) and is now administered by the MTConnect Institute. See the mtconnect.org web site for more detailed information.

This section will only cover the deployment issues related to Windows installation of MTConnect. MTConnect deploys either a 32 bit or 64 bit service to read devices data and then provides a Web server to format the data into a standard XML. MTConnect installation has these basic requirements:

1. Install Microsoft Visual C++ and other dll
2. Install 64-bit agents and adapters as services: in NSIS distinguish install location
   1. InstallDir $PROGRAMFILES32\MTConnect\MTConnectAgent
   2. InstallDir $PROGRAMFILES64\MTConnect\MTConnectAgent
3. Configure Adapters to modify/create agent.cfg
4. Generate Devices.xml for an agent, based on adapter configuration
5. Configure Agent service name, port and add adapters (minimally name, type, ip, socket).
6. Reset file permissions of installed files, e.g., agent.cfg.agent.log so they can be written to and save.

# NSIS Background

NSIS stands for NullSoft Installation Software. NSIS is currently a SourceForge project that is freely downloaded. NSIS is a script language for building a “Wizard like” installation GUI. You program a script, compile it to generate install and uninstall exe for Windows. NSIS has been around for 15-20 years. Unfortunately, there is a “modern” interface and an old interface, and lots of useful bits of software technology thrown in.

Please read <http://nsis.sourceforge.net/Docs/> for detailed online documentation of NSIS

Useful web sites:

NSIS Modern UI: <http://nsis.sourceforge.net/Docs/Modern%20UI%202/Readme.html>   
Code Project: <http://www.codeproject.com/Articles/24187/Creating-an-Installer>   
NSIS nsDialogs Plug-in: <http://nsis.sourceforge.net/nightly/Docs/nsDialogs/Readme.html>   
Changing Sheet buttons: <http://nsis.sourceforge.net/Buttons_Header>

Useful NSIS headers: <http://nsis.sourceforge.net/Docs/AppendixE.html>

UltraModernUI <http://ultramodernui.sourceforge.net/>

## NSIS Custom Dialog Pages

Custom dialogs are critical for configuring the agent and then adding adapter specifics (e.g., IP address, TCP socket) from which to read data. There are many software packages to assist developers in building custom dialogs. The package used in the deployment described is the freeware Sourceforge NSIS Dialog Designer, which is a graphical designer for NSIS nsDialog setup dialogs. You can download the page from the web: http://nsis.sourceforge.net/NSIS\_Dialog\_Designer.

The Sourceforge NSIS Dialog Designer package provides a visual studio like canvas on which to place GUI widgets, and will generate an include file that NSIS uses. For example, after building an Adapter configuration screen, the include portion of the NSIS script is:

!include "MTCAdapterConfig.nsdinc"

Then within the page deployment layout the format for deployment is:

Page custom fnc\_MTCAdapterConfig\_Show AdapterConfigPageLeave

The “Page custom” notifies NSIS that a custom page is to be displayed, with a create/show beginning function and a leave page when done.

The Sourceforge NSIS Dialog Designer allows “Code insertion” and notification callback on events, such as:change due to navigation, or button click.

## Remapping Sheet Navigation Buttons

The standard buttons for the Install sheet layout are: Back, Next, and Cancel. Depending on the type of install sheet, various combinations are enabled, or renamed. Since one or more Adapters can be inserted and configured, it was easiest to remap the buttons. For the GUI sheet giving Adapter Configuration, Back became Apply and Next became Done.

To achieve remapping of the Sheet traverse button, the Script (actually function call) “Call ConfigPreCreate” was inserted to remap the button. Figure 1 shows the Sourceforge NSIS Dialog Designer display, and the use of a script to remap the sheet navigation buttons.

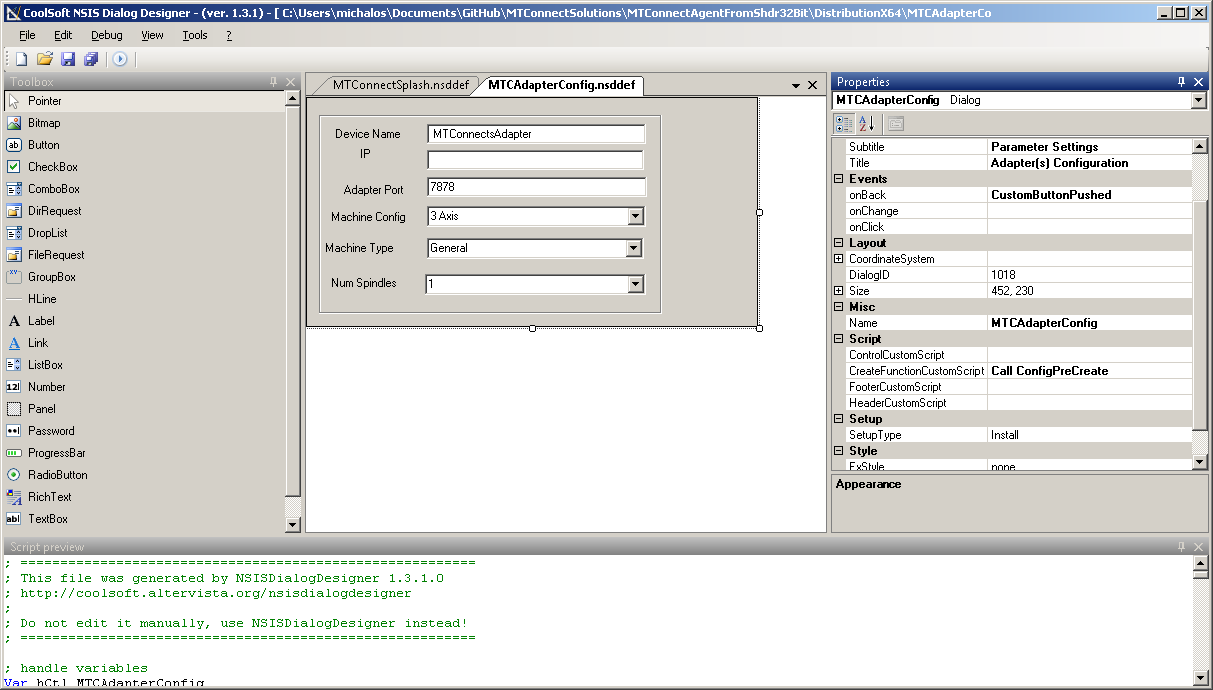


Figure 1 Sourceforge NSIS Dialog Designer Visual Designer Windows

The actual ConfigPreCreate function requires NSIS sheet remapping and requires an understanding of how Windows does GUI widget creation, handling and access methods.

Function ConfigPreCreate

GetDlgItem $8 $HWNDPARENT 1 ; next, close == 1

SendMessage $8 ${WM\_SETTEXT} 0 "STR:&Done"

EnableWindow $8 1

ShowWindow $8 1

GetDlgItem $8 $HWNDPARENT 2 ; cancel == 2

SendMessage $8 ${WM\_SETTEXT} 0 "STR:&Cancel"  
EableWindow $8 1

ShowWindow $8 1

GetDlgItem $8 $HWNDPARENT 3 ; back==3

SendMessage $8 ${WM\_SETTEXT} 0 "STR:&Apply"

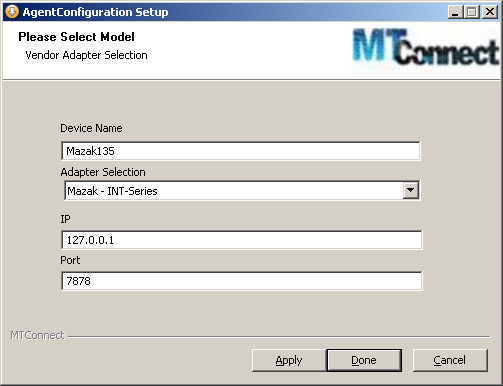
EnableWindow $8 1

ShowWindow $8 1

FunctionEnd

Above the use of $8 is a general variable and is part of a set of NSIS predefined variables (e.g., $0-$9), and holds values globally. In the statement “GetDlgItem $8 $HWNDPARENT 1” the GetDlgItem retrieves the dialog item handle (button) within the parent dialog Window. Basically, Windows uses “Handles” to all windows on the screen. The main handle ($HWNDPARENT) defines the window handle to the NSIS sheet. GetDlgItem corresponds to the windows function to get the window handle to the dialog item, in this case, either 1, 2 or 3. The handle 1 corresponds to the Next button, which is remapped to Done. SendMessage $8 ${WM\_SETTEXT} 0 "STR:&Done" handles setting the Button text, using the Windows SendMessage function, the handle contained in variable $8, the NSIS variable $(WM\_SETTEXT)containing the Windows message number corresponding to WM\_SETTEXT, 0 for the WPARAM and the string “Done” for the LPARAM, which is passed as "STR:&Done". The handle 2 corresponds to the Cancel button, which is not remapped. And the handle 3 corresponds to the Back button, which is remapped to Apply.

This gives a sheet like:



Normally once you click Back (in this case Apply), you automatically navigate to the previous sheet. However, in this case, we only want to save the Adapter configuration and then reset the configuration page so the user can input another Adapter configuration. To allow multiple Adapters to be defined, the Function CustomButtonPushed was inserted into the OnBack Event slot (see below).

The CustomButtonPushed is a NSIS Function that records the current adapter configuration, resets the dialog to an initial state, and then invokes Abort, which informs NSIS not to automatically traverse to the previous sheet. AddAdapterConfig and ResetAdapterConfig are both NSIS functions, so according to Function syntax, must be invoked by a “Call”. Note, NSIS is case sensitive, so “call” will not work.

Function CustomButtonPushed ; back button of adapter config page

Call AddAdapterConfig

Call ResetAdapterConfig

Abort

FunctionEnd

However, since we are adding vendor based templates, we are moving back a page. So instead of clearing and staying on the same sheet, the Adapter configuration moves “back” two pages. (See code below in place of “Abort”.)

StrCpy $R9 "2" ;Relative page number. See below.

Call RelGotoPage

Two is a relative number, which is set to the local variable $R9 and then the Function RelGotoPage is called. The RelGotoPage uses the Microsoft windows function SendMessage to the parent window handle ($HWNDPARENT) to move "$R9” relative pages.

Function RelGotoPage

IntCmp $R9 0 0 Move Move

StrCmp $R9 "X" 0 Move

Move:

SendMessage $HWNDPARENT "0x408" "$R9" ""

FunctionEnd

IntCmp is defined as:

IntCmp val1 val2 jump\_if\_equal [jump\_if\_val1\_less] [jump\_if\_val1\_more]

IntCmp compares two integers val1 and val2. If val1 and val2 are equal, [Gotos](http://nsis.sourceforge.net/Reference/Goto) jump\_if\_equal, otherwise if val1 < val2, [Gotos](http://nsis.sourceforge.net/Reference/Goto) jump\_if\_val1\_less, otherwise if val1 > val2, [Gotos](http://nsis.sourceforge.net/Reference/Goto) jump\_if\_val1\_more.

While StrCmp is defined as:

StrCmp str1 str2 jump\_if\_equal [jump\_if\_not\_equal]

StrComp compares (case insensitively) str1 to str2. If str1 and str2 are equal, the function performs g[oto](http://nsis.sourceforge.net/Reference/Goto) jump\_if\_equal, otherwise performs g[oto](http://nsis.sourceforge.net/Reference/Goto) jump\_if\_not\_equal.

If the “Done” button (remapped from Next) is pushed, then the leave callback Function is invoked, which was specified as “AdapterConfigPageLeave”:

Page custom fnc\_MTCAdapterConfig\_Show AdapterConfigPageLeave

“AdapterConfigPageLeave” is a function which reads the Adapter configuration, generates the Adapter configuration used in the Agent.cfg file, and NSIS traverses to the next screen (if one).

Function AdapterConfigPageLeave

Call AddAdapterConfig

FunctionEnd

## NSISList utility

The NSISList utility was ultimately used to store lines in a file, since NSIS strings are only 1024 bytes in length, so concatenating long string (such as XML adapter definition used in devices.xml) was out of the question. Thankfully, the NSISList plugin existed, so each file line was assigned to one list item, so a file could be created and the list could be as large as necessary (at least within our small testing boundaries).

The AddAdapterConfig function was responsible for reading the adapter settings (after the vendor and machine type have been configured.) and then generating the Adapter configuration for Agent.cfg and Devices.xml. The AgentCfgList was a NISTList facility used to save lines in the file Agent.cfg. For each adapter, new lines were added to AgentCfgList that would later be written out at once to a file.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Function AddAdapterConfig

;Read Textboxes values

${NSD\_GetText} $hCtl\_MTCAdapterSelection\_TextBox2 $AdapterPort ; adapter port, e.g., 7878

${NSD\_GetText} $hCtl\_MTCAdapterSelection\_TextBox1 $AdapterIP ; adapter IP e.g., 127.0.0.1

${NSD\_GetText} $hCtl\_MTCAdapterSelection\_TextBox3 $AdapterName ; adaptername

${NSD\_GetText} $hCtl\_MTCAdapterSelection\_DropList1 $6 ; cnc type general

;; This creates the agent.cfg entry

${List.Add} AgentCfgList "$\t$AdapterName$\r$\n"

${List.Add} AgentCfgList "$\t{$\r$\n"

${List.Add} AgentCfgList "$\t$\tHost=$AdapterIP$\r$\n"

${List.Add} AgentCfgList "$\t$\tPort=$AdapterPort$\r$\n"

${List.Add} AgentCfgList "$\t$\tDeviceType=$6$\r$\n"

${List.Add} AgentCfgList "$\t}$\r$\n"

${GenerateDevicesXML} $6 $AdapterName

FunctionEnd

The ${GenerateDevicesXML} $6 $AdapterName was responsible for generating a <Device> section corresponding to the adapter, within the “Devices.xml” file. The NSIS variable $6 hold the vendor template “Filename” minus the “.txt” extension. There are only a few Vendor MTConnect templates that are used to create MTConnect data configuration portions. To develop a Devices.xml specification, each within device template file the string “####” is replaced with the actual AdapterName (i.e., the Device name). It is assumed the Device name is unique or the ids will not be unique, which is an XML error.

NSIS code was fund to replace substrings within a string (see <http://nsis.sourceforge.net/StrReplace_v4> for details of code). What is of note, is that parameters cannot be passed to a Function, but instead must use a Macro. So this is a little convoluted, but you get used to it.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; StrReplace

;; ${StrReplace} "C:\my documents\" "\" "//"

;; MessageBox MB\_OK $0

!define StrReplace '!insertmacro "\_StrReplaceConstructor"'

!macro \_StrReplaceConstructor ORIGINAL\_STRING TO\_REPLACE REPLACE\_BY

Push "${ORIGINAL\_STRING}"

Push "${TO\_REPLACE}"

Push "${REPLACE\_BY}"

Call StrRep

Pop $0

!macroend

Due to the string length limit (1024), the function AddAdapterConfig uses the List functionality to store file lines. First, AddAdapterConfig uses windows functions to read the GUI widget settings of interest (3 text boxes and 3 selections from a drop down list). The function FcnGenerateOneDevicesXML reads eachs read each line of the template device file. For each line of the Device template file, the “####” string is replaced. Finally, the modified string is appended to the DevicesXmlList list.

Function FcnGenerateOneDevicesXML

Var /GLOBAL DeviceName

Var /GLOBAL FileTitle

Var /GLOBAL Line

Pop $DeviceName

Pop $FileTitle

FileOpen $4 "$INSTDIR\Devices\$AdapterVendor\$FileTitle.txt" r

Loop:

IfErrors done

FileRead $4 $Line

DetailPrint $Line

${StrReplace} $Line "####" "$DeviceName"

StrCpy $Line $0

${StrReplace} $Line "NNNNNN" "$DeviceName"

StrCpy $Line $0

${List.Add} DevicesXmlList $Line

IfErrors 0 Loop   
 done:

FileClose $4

FunctionEnd

After all the adapters are configured, the

## Installation for Image Handling – please read cautionary note at end

One would like to have “brand” images and icons associated with the installation. NSIS can do this, but it is not trivial. You cannot use images that have not been installed in custom pages, or they did not work for me. To make image branding, this web site offered guidance that worked, <http://nsis.sourceforge.net/%22Orange%22_Modern_UI_Theme>

The web site contained an example that worked and was modified for the MTConnect image branding:

MUI Settings / Icons

!define MUI\_ICON "${NSISDIR}\Contrib\Graphics\Icons\orange-install-nsis.ico"

!define MUI\_UNICON "${NSISDIR}\Contrib\Graphics\Icons\orange-uninstall-nsis.ico"

; MUI Settings / Header

!define MUI\_HEADERIMAGE

!define MUI\_HEADERIMAGE\_RIGHT

!define MUI\_HEADERIMAGE\_BITMAP "${NSISDIR}\Contrib\Graphics\Header\orange-r-nsis.bmp"

!define MUI\_HEADERIMAGE\_UNBITMAP "${NSISDIR}\Contrib\Graphics\Header\orange-uninstall-r-nsis.bmp"

; MUI Settings / Wizard

!define MUI\_WELCOMEFINISHPAGE\_BITMAP "${NSISDIR}\Contrib\Graphics\Wizard\orange-nsis.bmp"

!define MUI\_UNWELCOMEFINISHPAGE\_BITMAP "${NSISDIR}\Contrib\Graphics\Wizard\orange-uninstall-nsis.bmp"

The issue "no branding image found" message arises and is related to:

MUI\_LANGUAGE macro(s) have to come after the MUI\_PAGE\_\* macros in the source file

From: <http://stackoverflow.com/questions/1341321/missing-welcome-image-in-nsis-mui2>

**Not a trivial problem**, but duly noted.

## Using NSIS to detect windows 32bit or 64 bit platform

Currently, PC platforms are either 64-bit or 32-bit architectures. (Note discussion that follows assumes Intel x86 architectures – either 32 or 64 bit.) Generally, if you are using Windows XP you have a 32-bit platform and if you Windows 7 you have a 64-bit platform. Often, a 32-bit executable will run on either a 32 or 64 bit architecture with no problem This was found not to be the case with the MTConnect Agent , which is platform specific. So, a 32 bit exe and a 64 bit exe are generated and must be distributed. Fortunately, NSIS makes determining the current architecture (322 or 64 bit) and then selecting the correct installation folder and exe possible. Using the x64.nsh NSIS “plug-in” from <http://nsis.sourceforge.net/Include/x64.nsh> as explained in <http://stackoverflow.com/questions/13229212/how-to-detect-windows-32bit-or-64-bit-using-nsis-script>. This code used boils down to:

!include x64.nsh

!include "logiclib.nsh"

And use the “if” statement , you must include logiclib.nsh to use the $(if) conditional:

${If} ${RunningX64}

# 64 bit code

${Else}

# 32 bit code

${EndIf}

Using this logic x32 and x64 folders were created that contained MTConnect Agent executables for architecture that are loaded depending on the given platform. Not only is the Agent to 32 or 64 bit architectures, but the Microsoft visual C++ dll install (vc\_redist.exe) are directed to 32 or 64 architectures.

${If} ${RunningX64}

File "x64\agent.exe"

File "x64\msvcrt.dll"

File "x64\msvcr100.dll"

${Else}

File "x32\agent.exe"

File "x32\msvcrt.dll"

File "x32\msvcr100.dll"

${EndIf}

Of note, the Agent may use some functionality from the Microsoft Visual C++ (MSVC) compiler, which is encapsulated in a dll. If this dll is not loaded onto the system, the agent will fail to load and run and may report a missing dll. So, the specific 32 or 64 bit MSVC dll is loaded in the executable folder (so it can be found first by the agent.exe bootstrapper). For agents compiled with MSVC 2010, the msvcr100.dll must be loaded. For agents compiled with MSVC 2008, the msvcr90.dll must be loaded.

# Load files from a directory into InstallOptions drop down

See: <http://nsis.sourceforge.net/Load_files_from_a_directory_into_InstallOptions_drop_down>

Currently, the device templates are organized by vendors, machine types (machine tool, turning) and then by model. It is assumed at this time that the model describes the number of axes in the machine configuration. This may not be valid, but works as a first attempt.

An Adapter model selection is used by the Agent.cfg file and the Devices.xml file to specify the adapters and their configurations for the Agent. The Agent.cfg specifies the device name, IP port, and the TCP socket to read the SHDR. Of note the Adapter model type is saved in the Agent.cfg file, so that it can be loaded and modified by an application.

In order to select the correct Adapter model setting, lists are created to handle Agent.cfg (AgentCfgList) and the Devices.xml (DevicesXmlList) file string contents. Lists are used since individual lines are less that 1024 characters in length, however, since the total files length may exceed 1024 bytes, one string cannot handle the entire configuraiton.

Function .onInit

${List.Create} DevicesXmlList ; Create a list for Devices,Xml output

${List.Create} AgentCfgList ; Create a liste for Agent.cfg output

Next the custom sheet function AdapterVendorPageLeave is a sheet callback function that reads the vendor and machine type in order to determine the vendor model type.

Function AdapterVendorPageLeave

;Read Sheet values

${NSD\_GetText} $hCtl\_MTCAdapterInstall\_ComboBox2 $AdapterVendor

${NSD\_GetText} $hCtl\_MTCAdapterInstall\_ComboBox3 $MachineType

FunctionEnd

Next, given a vendor and machine type the names of the files in the folder $INSTDIR\Devices\$AdapterVendor\$MachineType are read, and then appended to the $hCtl\_MTCAdapterSelection\_DropList1 drop down list. The user can then select a model based this list.

Function FindDevicesFiles

ClearErrors

SendMessage $hCtl\_MTCAdapterSelection\_DropList1 ${CB\_RESETCONTENT} 0 0

FindFirst $R0 $R1 "$INSTDIR\Devices\$AdapterVendor\\*.txt"

Loop\_jobs:

IfErrors Loop\_jobs\_end

${StrReplace} $R1 ".txt" ""

SendMessage $hCtl\_MTCAdapterSelection\_DropList1 ${CB\_ADDSTRING} 0 "STR:$0"; "$R1"

FindNext $R0 $R1

IfErrors 0 Loop\_jobs

Loop\_jobs\_end:

FunctionEnd

Writing out the Agent.cfg file, then constitutes only opening the file, giving the file handle into

FileOpen $1 "$INSTDIR\Agent.cfg" "w"

Now, the count of lines in the Agent.cfg file is stored in NSIS predefined global variable $2, and 1 is substracted from this count, since the for loop count is zero based, and the list count is 1 based.

${List.Count} $2 AgentCfgList

IntOp $2 $2 - 1

And then the for loop to writes each item in the list to the file as given by handle in predefined variable $1. Each line is indexed by $i (defined earlier as a var) from the AgentCfgList and then is extract into NSIS predefined variable $3. Finally fetched line “$3” is written to the file defined by the handle given in “$1”, which gives:

${For} $i 0 $2

${List.Get} $3 AgentCfgList $i

FileWrite $1 "$3"

${Next}

Putting all this NSIS code together gives:

Function WriteAgentCfgFile

FileOpen $1 "$INSTDIR\Agent.cfg" "w"

${List.Count} $2 AgentCfgList

IntOp $2 $2 - 1

${For} $i 0 $2

${List.Get} $3 AgentCfgList $i

FileWrite $1 "$3"

${Next}

FileClose $1

FunctionEnd

## Problem: How to reassign full read/write control to all file?

A 32 windows exe was written that reassigns file permissions using the executable app: SetFilePermission. This is very useful for Windows 7 PCs, since these platforms forbid write access to the install directory after the install. Instead windows expects installed programs to write to the users app data folder. Since the app is really an service exe, the location of the debug files (agent.log, etc.) is confusing since the log files are hard to find. Further, hand configuration is still an option (so more adapters may be added to the agent), and to save the agent.cfg file, the read/write permissions must be relaxed to allow this.

SetFilePermission can either loosed permissions (so everyone can read/write) for a folder or for an individual file. Thus the syntax of the SetFilePermission application is:

SetFilePermission [Folderpath|Filepath]

In order for NSIS to run SetFilePermission, the ExecWait facility is used. ExecWait execute the called string, be it some application, console or file, and waits till the process to exit. Below, the last $0 will be loaded with the return value from SetFilePermission.

ExecWait '"$INSTDIR\SetFilePermission.exe" "$INSTDIR"' $0.

Because the file Agent.cfg is deleted and then overwriten with a fresh adapter(s) configuration and agent configuration, at the end the installation Agent.cfg needs to be reset to allow r/w for everyone permission, so the Agent.cfg file is sent to SetFilePermission.exe. The function .onGUIEnd is an NSIS predefined event callback function, done after the installation gui has completed running.

Function .onGUIEnd

Call GenerateAgentCfg

ExecWait '"$INSTDIR\SetFilePermission.exe" "$INSTDIR\Agent.cfg"' $0

FunctionEnd

## Issue: Can NSIS support a drop down menu for selection of Device Types?

Not that I could find. This would be desirable for selecting adapters in a drop down menu based on vendor, machine type, model, etc. However, since drop down menus are difficult to use on small screen mobile devices, their use has declined, and NSIS GUI development tools don’t seem to support the facility.

## Issue: You would like to disable the Back button.

You can disable the button with the EnableWindow command in the show callback function of the Finish page.

GetDlGItem $0 $HWNDPARENT 3

EnableWindow $0 0

## Issue: Installing files recursively, and excluding directories

File /r /x dir-to-exclude-1

## Issue: Embedded Quotation Marks

$\"$\" and not $\$\"

## Issue: Set File Permission Code

Below is the code that was found on the internet that sets file permissions correctly to everyone R/W for a file. Other code does finding files in a folder, but this code is not special.

#include "Aclapi.h"

void CFilePermission::SetFilePermission(std::string szFilename)

{

LPCTSTR FileName= szFilename.c\_str();

DbgOut(StdStringFormat("SetFilePermission %s", FileName).c\_str());

PSID pEveryoneSID = NULL;

BOOL a= TRUE;

BOOL b= TRUE;

BOOL c= TRUE;

DWORD d= 0;

DWORD ret = 0;

PACL pACL = NULL;

EXPLICIT\_ACCESS ea[1];

SID\_IDENTIFIER\_AUTHORITY SIDAuthWorld = SECURITY\_WORLD\_SID\_AUTHORITY;

// Create a well-known SID for the Everyone group.

a = AllocateAndInitializeSid(&SIDAuthWorld, 1,

SECURITY\_WORLD\_RID,

0, 0, 0, 0, 0, 0, 0,

&pEveryoneSID);

if(!a)

DbgOut("SID initialization failed");

// Initialize an EXPLICIT\_ACCESS structure for an ACE.

// The ACE will allow Everyone read access to the key.

ZeroMemory(&ea, 1 \* sizeof(EXPLICIT\_ACCESS));

ea[0].grfAccessPermissions = 0xFFFFFFFF;

ea[0].grfAccessMode = GRANT\_ACCESS;

ea[0].grfInheritance= NO\_INHERITANCE;

ea[0].Trustee.TrusteeForm = TRUSTEE\_IS\_SID;

ea[0].Trustee.TrusteeType = TRUSTEE\_IS\_WELL\_KNOWN\_GROUP;

ea[0].Trustee.ptstrName = (LPTSTR) pEveryoneSID;

// Create a new ACL that contains the new ACEs.

ret = SetEntriesInAcl(1, ea, NULL, &pACL);

if(ret != ERROR\_SUCCESS)

DbgOut("ACL entry failed");

// Initialize a security descriptor.

PSECURITY\_DESCRIPTOR pSD = (PSECURITY\_DESCRIPTOR) LocalAlloc(LPTR,

SECURITY\_DESCRIPTOR\_MIN\_LENGTH);

b = InitializeSecurityDescriptor(pSD,SECURITY\_DESCRIPTOR\_REVISION);

if(!b)

DbgOutError(::GetLastError(),"Security Descriptor initialization failed");

// Add the ACL to the security descriptor.

c = SetSecurityDescriptorDacl(pSD,

TRUE, // bDaclPresent flag

pACL,

FALSE); // not a default DACL

// If the function SetSecurityDescriptorDacl succeeds, the function returns nonzero.

if(c==0)

{

DbgOutError(::GetLastError(),"SetSecurityDescriptorDacl failed");

}

//Change the security attributes

d = SetFileSecurity(FileName, DACL\_SECURITY\_INFORMATION, pSD);

// If the SetFileSecurity function succeeds, the function returns nonzero.

if(d == 0)

DbgOutError(::GetLastError(),"SetFileSecurity failed");

if (pEveryoneSID)

FreeSid(pEveryoneSID);

if (pACL)

LocalFree(pACL);

if (pSD)

LocalFree(pSD);

}

## Issue: Dialog with URL Link

URL link can be part of the custom dialog but need the old Modern interface. This web page describes a plug-in <http://nsis.sourceforge.net/Linker_plug-in> to handle link (using labels and subwindows).