AGFM MTConnect Agent Deployment

August 17, 2015

This document presents a brief background on the mechanics of the MTConnect agent for the AGFM CNC. This document assumes the reader is familiar with MTConnect operation, and for deeper explanation of MTConnect, please refer to MTConnect URL: <http://www.mtconnect.org/> for more information. This document concerns itself with how an embedded Adapter operates. One or more embedded adapters read a configuration file, and then read a remote AGFM log files as logged from an AGFM CNC device. AGFM provides a log file mechanism, that must be enabled, and then a file will be created. This AGFM log file must be shared in Windows so the MTConnect embedded adapter can read the file, reinterpret the contents into MTConnect meaningful terminology, and then report this information to a canned MTConnect Agent (circa version 1.2), which handles management of devices(s) data, the web service, etc. The AGFM log file is a Comma Separated File (CSV) with the semicolon “;” as the delimiter between log fields on a single line.

# Background

MTConnect is a new standard developed to facilitate the exchange of data on the manufacturing floor. The MTConnect open specification provides for cost effective data acquisition on the manufacturing floor for machine tools and related devices. MTConnect is based upon prevalent Web technology including XML and HTTP. Figure 1 shows the MT Connect architecture. An “MTConnect Device” is a piece of equipment – a Mazak, DST, Echospeed or other machine tool, which (optionally) includes an MTConnect Adapter so that we can get data from it. The “Agent” is a process that acts as a “bridge” between a device and a factory “Client Application”. To learn more about MTConnect visit: <http://www.mtconnect.org/>

Figure 1 shows a typical MTConnect AGFM Logging Agent system architecture (with only one AGFM device). Communication between two Windows PCs is assumed - one containing the AGFM log file and the other PC communicating over Ethernet to read the AGFM log file.



Figure MT Connect Architecture

# Configuration

## Ini File Terminology

The AGFM log file agent relies on an INI configuration file that is not typical of normal MTConnect installations. The INI file format is an informal standard for configuration. 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 AGFM 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

## Agfm Ini File Configuration

The AGFM configuration file is Config.ini. The file Config.ini in the same folder as the executable folder contains the configuration information.

The [GLOBALS] section contains some basic Agent/Adapter parameters to configure the MTConnect Agent, and the embedded AGFM logging adapter. Below is a sample [GLOBALS] section in the Config.ini INI file.

[GLOBALS]

HttpPort=5000

ServiceName=AgfmAgent

ResetAtMidnite=false

Debug=0

MTConnectDevice=M1

logging\_level=FATAL

QueryServer=10000

ServerRate=5000

The MTConnect Agent configuration parameters (HttpPort, ServiceName) will be covered later.

* The tag ResetAtMidnite is a Boolean which tells the agent whether it needs the agent to reset at midnite every night, which is useful if any underlying software leaks memory, is flaky, etc. A reset at midnight cures a lot of software ills.
* The tag Debug is a flag for the embedded adapter and ranges from 0 (none) to 5 (full). This debug information is reported into the file “debug.txt” located in the same folder as the executable.
* The tag logging\_level refers to logging for the MTConnect Agent, and is usually . Logging information from the MTConnect Agent is stored in the file Agent.log, also collocated with in the executable folder.
* QueryServer rate parameter sets up how long to query a remote device if it is down (for example the computer where the log file resides is turned off.) The QueryServer rate is in milliseconds, so 10000 is 10 seconds.
* ServerRate parameter is the wait between embedded adapter reads of the log file. Again, the time representation is in milliseconds. The QueryServer and the ServerRate parameters are the default parameters for any device that does not specify a value for these parameter.

MTConnect requires two configuration files: agent.cfg (yaml format) and Devices.xml (MTConnect XSD format). To satisfy the Agent.cfg need, the Config.ini contains the following:

[GLOBALS]

ServiceName=AgfmAgent

which provides the service name of the agent that will be created within the Windows OS Service Control Manager. Also the HTTP port that the agent listens to provide the MTConnect web service is also defined in the GLOBALS section:

[GLOBALS]

HttpPort=5000

This means to test if the MTConnect Agent is operating the following URL must be used:

http://127.0.0.1:5000/current

This test will provide all the current data for any MTConnect device under control by this MTConnect Agent.

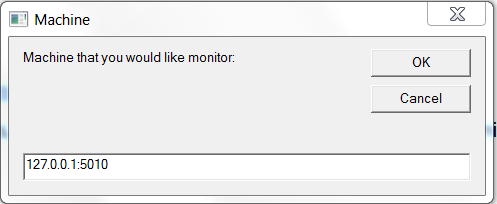
# Installation AGFM Log Agent

Important notes, the Service in the cannot be removed unless Uninstall.vbs is (right click) Runas Administrator.

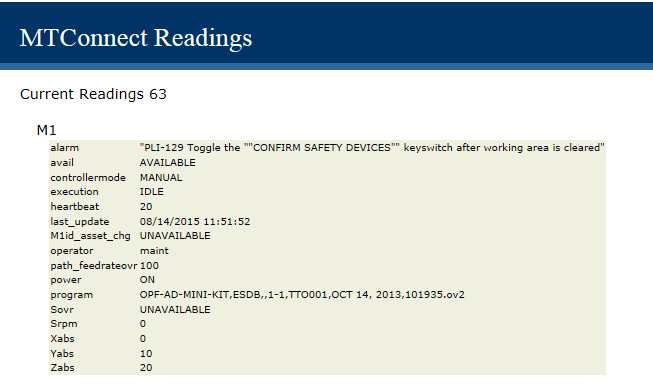
If the file does not have today or yesterday’s date, the Agent XML will display unavailable.

Agent reads the config.ini to write a new Devices.xml – every time. It is hard coded this way.

To debug the Agent, run the vb script: MTConnectPage-1.4.vbs and input a matching URL with port



Which should then display something similar to this:



## Configuring MTConnect Log Devices

Now configuration of the Devices.xml file is done with the Config.ini file. The AgfmLogFile Agent looks at the “MTConnectDevice” name tag to see how many devices with accompanying sections are defined. (The device name must be unique, but no check is done.)

[GLOBALS]

MTConnectDevice=M1

Thus M1 has a unique section in the Config.ini file that defines all the necessary configuration parameters required. For actual deployment in a shop environment, only the ini file tag “ProductionLog” is important and must be instantiated, because the embedded MTConnect “Adapter” must know where the log file is located.

[M1]

ProductionLog=C:\Users\michalos\Documents\GitHub\Agents\AgfmMTConnectAgent\AgfmAgent\x64\Debug\ProductionLog.csv

logging\_level=ERROR

Simulation=1

LastLogDate=10/9/2013

QueryServer=20000

ServerRate=2000

Typically, a UNC file name is used to describe the file for the production log. In the example above used for testing, a local file with simululation flags set is shown.

In the UNC Name Syntax UNC names identify network resources using a specific notation. UNC names consist of three parts - a server name, a share name, and an optional file path. These three elements are combined using backslashes as follows:

\\server\share\file\_path

The server portion of a UNC name references the strings maintained by a network naming service such as DNS or WINS. Server names are “computer names”. The share portion of a UNC name references a label created by an administrator or, in some cases, within the operating system. In most versions of Microsoft Windows, for example, the built-in share name admin$ refers to the root directory of the operating system installation (usually C:\WINNT or C:\WINDOWS). The file path portion of a UNC name references the local subdirectories beneath the share point. For example:

\\grandfloria\temp (to reach C:\temp on the computer grandfloria)

In this example, the file temp must be shared to allow remote access for MTConnect to read the file. Also, file permissions must allow read access to this file. This may require administrator privileges in order to perform the file read. The embedded MTConnect adapter only reads the file, it does not write, delete or modify in any manner the contents of the log file. It is on the onus of the logging service of the CNC to modify the log file. Excessively large log files, will result in delays reading thru the log file to reach the latest reading (typically because we cannot be sure what, when is recorded in the log file.) The AGFM logging service shrinks the log file by 20% when it reaches a maximum size (10M), so this is not of major concern.

The QueryServer and the ServerRate parameters override the global definition, and define how long to wait between down devices and embedded adapter updates.

The ini file parameter Simulation determines if the log file is a simulation and contains old but useful data. If the simulation flag is true, one sets the parameter “LastLogDate” to the date one wants the simulated log file reading to start at. The format of the data is mm/dd/yyyy, for example, 10/9/2013.

# Log file

The CNC6000 is capable of logging many events into a file for later use by production management to track the machine’s production capacity. This file, by default, is called PRODUCTION1.CSV and is stored in the CNC\USERDATA folder. The max file size of the CSV file is configured via GFMLogger.INI, using the key “MaxFileSize”. The default value for the file size is 10485760 bytes (or 10MB). When this limit is reached, the oldest 20% of the file will be cut off. The production log data is stored in CSV (Comma Separated Values) format. Each line in the log has the following entries:

* Timestamp (date and time)
* Name of the currently logged on user
* An Event ID number (see the following table for possible events)
* Short description of the Event (in defined language)
* Additional data depending on the event.

|  |  |  |
| --- | --- | --- |
| **EVENT ID** | **Logged Event** | **Event Description** |
| **0** | Program Loaded | Loaded Loading of a program to the CNC-control  Additional data:  - Name of the loaded program |
| **1** | Program Begin | When the operator start an automatic program  Additional data:  - Name of the currently loaded program  2 Program Restarted When the operator continuous an previously not finished program  Additional data:  - Name of the currently loaded program |
|  |  |  |
| **3** | Program End | After finishing a program  Additional data:  - Name of the currently loaded program  - Cutting length  - Total processing time between start end end of the program  - Cutting Time = G1  - Idle Time: Difference between Total and Cutting time |
| **4** | Program Abort | Abortion of a program  Additional Data:  - Name of the currently loaded program |
| **5** | Window Transfer | Documents the conveyor transfer when window cutting is used  Additional data:  - Name of the currently loaded program |
| **6** | FeedholdON | ON logs when feed hold is activated  Additional data:  - Name of the currently loaded program |
| **7** | Feedhold OFF | Additional data:  - Name of the currently loaded program |
| **8** | Log Begin | Start of the MMI |
| **9** | Log End | Closing of the MMI |
| **10** | Alarm | Active Error during a program run  Additional data:  - Alarm number and text in defined language |
| **11** | Override Value | Changes of Override value  Additional data:  - Name of the currently loaded program  - new Override value in percent |
| **12** | Tool Change | Logs the tool change  Additional data:  - Name of the currently loaded program  - number of the actual tool |

# AGFM Embedded Adapter C++ Code Explanation

The algorithm within the method Adapter::GatherDeviceData() is as follows. First, a UNC network shared file is provided within the Config.ini file to specify where the CNC event file is located for the adapter to read. Next, the UNC file is checked to see if it exists. If file is not found, an error code is generated and a message is returned from 'GatherDeviceData' signally no file.

if(!File.Exists(filename))

{

return FailWithMsg(E\_FAIL, StdStringFormat("UNC File %s not found for device %s\n", filename.c\_str(), \_device.c\_str())) ;

}

Next, if the file exists, the modification time is checked against the last modification time. This check will fail if the modification times (using the Microsoft date/time type COleDateTime types) are the same. If the check succeeds the next test is done, otherwise and error message is return from GatherDeviceData.

// Check mod time, if same, returns

modtime = File.GetFileModTime(filename);

if(lastmodtime==modtime)

return FailWithMsg(E\_PENDING, StdStringFormat("File %s Device %s modtime unchanged %s\n", filename.c\_str(), \_device.c\_str(), modtime.Format("%c"))) ;

Assuming success, the next check is to see if the file size is larger than the last file size. If the two file sizes (current and last update) are the same, then no new information is available and an error code is generated and message is returned signally file size failed. If the check succeeds the next test is done.

// Check file size, if same, returns

if(FAILED(File.Size(filename, filesize)))

return FailWithMsg(E\_FAIL, StdStringFormat("File %s Device %s filesize Failed\n", filename.c\_str(), \_device.c\_str()))

if(filesize==\_lastfilesize)

return FailWithMsg(E\_PENDING, StdStringFormat("File %s Device %s filesize:%d = lastfilesize:%d\n", filename.c\_str(), \_device.c\_str(),filesize,\_lastfilesize)) ;;

Next if the file size is checked to see if there is indeed a file size. If the file size is zero, then no information is available and an error code is generated and message is returned signally failed with file size zero. This concludes the file tests.

if(filesize==0)

return FailWithMsg(E\_PENDING, StdStringFormat("File %s Device %s filesize:%d = 0\n", filename.c\_str(), \_device.c\_str(),filesize)) ;

Next a std::string buffer is allocated according to the size of the file. We have previously saved the filesize, so we use this to create a std::string and then resize it according to the filesize.

long ulFileSize=filesize;

std::string data;

data.resize(ulFileSize);

Previously all other C++ read file mechanisms were attempted but did not work with Microsoft UNC path names. So instead, the suite of Mircrosoft Windows File operations was used. In this scenario, the first step is to CreateFile with parameterization that allows read/write/delete sharing and assumes there already is an existing file (as we tested for this previously). If the CreateFile fails, an error code and fail message is generated:

HANDLE hFile = CreateFile(filename.c\_str(),

GENERIC\_READ, // access (read) mode

FILE\_SHARE\_READ|FILE\_SHARE\_WRITE|FILE\_SHARE\_DELETE, // share mode

NULL, // pointer to security attributes

OPEN\_EXISTING, // how to create

FILE\_ATTRIBUTE\_NORMAL,// file attributes

NULL); // handle to file with attributes to copy

if (hFile == INVALID\_HANDLE\_VALUE) // did we succeed?

return FailWithMsg(E\_FAIL, StdStringFormat("INVALID\_HANDLE\_VALUE File \"%s\" for device %s\n", filename.c\_str(), \_device.c\_str())) ;

Next only the new part of the file (since the last time the file was read) is determined and used to set the file pointer (equivalent to fseek in C code). Thus, only the new part of the file is read. In this scenario, the lastfilesize is used as the pointer into the file from where to read. If never read, lastfilesize is zero, so it would read the entire file (but only once). Accordingly, ulFileSize which signifies the total file size and is used by a read buffer must be reduced by seek operation amount. If the SetFilePointer fails, an error code and fail message is generated.

if(filesize>\_lastfilesize && \_lastfilesize != 0 )

{

DWORD dwPtr = SetFilePointer(hFile, \_lastfilesize, NULL, FILE\_BEGIN) ;

if (dwPtr == INVALID\_SET\_FILE\_POINTER) // Test for failure

return FailWithMsg(E\_FAIL, StdStringFormat("INVALID\_SET\_FILE\_POINTER File \"%s\" Error = %x for device %s\n", filename.c\_str(), GetLastError() , \_device.c\_str())) ;

ulFileSize=filesize-\_lastfilesize;

}

Now, the file can be read into the buffer provided (which will be released upon exist from this function so there will be no memory leaks). The buffer to store the ReadFile is pAddr and is a pointer to the allocated std::string data[0] buffer. If the CreateFile fails, an error code and fail message is generated. The variable dwNewSize should match the expected Read file size.

DWORD dwNewSize;

void \* pAddr = &data[0];

ReadFile(hFile, pAddr, ulFileSize, &dwNewSize, NULL );

CloseHandle(hFile);

Next, the last line of the file is extracted (ignoring all the potential updates in between). This is done by moving the buffer from the end of file backward until a '\n' is found signaling the end of previous line. Originally, we must skip the line feed and carriage return so the final file contents can be read into the std::string sentence. Each character is prepended onto the std::string sentence