

Windows Cloud Server Utility Documentation

OneBMCTest (BMC Test Tool)

Date: November 20, 2017  
Version: V2.00

**RESTRICTED**

THIS DOCUMENT AND ANY ASSOCIATED DATA CONTAIN RESTRICTED INFORMATION THAT IS MICROSOFT PROPERTY. DO NOT DISCLOSE TO OR DUPLICATE FOR OTHERS EXCEPT AS AUTHORIZED BY MICROSOFT CORPORATION.

|  |  |
| --- | --- |
| **Revision/ Date** | **Notes** |
| **V1.00** **5/9/16** | Initial document |
| **V1.01**  **5/19/16** | Added setup instructions for package required when interfacing via KCS  Added description for IpmiUtil.RunIpmiUtil function that can be optionally used in Test Scripts  Added section “Xml Batch File” with subsections |
| **V1.02**  **6/3/16** | Removed VC++ 2015 dependency and added additional dependencies |
| **V1.03**  **6/27/16** | Modified Setup section due to inclusion of setup packages |
| **V1.04**  **9/6/16** |  |
| **V1.05**  **12/20/16** | Modified documentation to account for additional BMC platform support (ie J2010) |
| **V2.00**  **11/20/17** | Modified all references to PyTestUtil and replaced with OneBMCTest |

Contents

[1 Overview 4](#_Toc470015532)

[2 Setup 4](#_Toc470015533)

[1 Pre-requisites 4](#_Toc470015534)

[2 Setup Procedures 5](#_Toc470015535)

[3 Running the Utility 5](#_Toc470015536)

[1 Quick-start 5](#_Toc470015537)

[2 Help text 7](#_Toc470015538)

[3 Syntax 7](#_Toc470015539)

[4 Adding Test Scripts 9](#_Toc470015540)

[1 Overview 9](#_Toc470015541)

[2 Creating a Test Script 9](#_Toc470015542)

[3 Adding a Test Script 13](#_Toc470015543)

[5 Xml Batch File 14](#_Toc470015544)

[1 Overview 14](#_Toc470015545)

[2 Modifying the Batch File 14](#_Toc470015546)

# Overview

OneBMCTest is a test utility to run functional and stress tests on the BMC.

The utility includes the following features:

* Command-line interface
* Supports both IPMI over LAN+ and KCS when interfacing with the BMC
* Supports REST API calls to J2010
* Can be run on Windows and Linux
* Includes batch file support to specify tests to be run
* Allows user to send email with full results
* Includes test scripts that can be expanded and new scripts can be added without changing any core utility code.

# Setup

## Pre-requisites

1. OneBMCTest tool package
2. Windows or Linux operating system with Python 2.7 installed
   1. NOTE: on Windows, add the installed Python directory to the PATH environment variable
   2. Python 2.7.9 install package for Windows 64-bit provided (Located in folder ./install as “python-2.7.9.amd64.msi”)
3. If running OneBMCTest via KCS, the following package may need to be installed:
   1. Visual C++ Redistributable for Visual Studio 2013 (32-bit)
      1. Link: <https://www.microsoft.com/en-us/download/details.aspx?id=40784>
      2. Setup package located in folder ./install as “vcredist\_x86.exe”
4. In order to run J2010 or AC power cycling tests, the “requests” module for Python needs to be installed:
   1. Link: <http://docs.python-requests.org/en/master/user/install/#install>
5. In order to run SSH or SFTP tests, the “paramiko” module for Python needs to be installed:
   1. Link: <http://www.paramiko.org/installing.html>
6. In order to generate the output Excel file, the python module XlsxWriter is required.

To install the XlsxWriter module:

$ sudo pip install XlsxWriter

(For Windows, ‘sudo’ is omitted).

1. Command-line interface (ie Powershell, Windows Command Prompt, Bash, etc.)
2. Remote machine connected to the same network as the computer server blade under test, or a compute server blade
3. When testing via IPMI over LAN+, the following pre-requisites may be needed but not required:
   1. 1 Gbps Ethernet switch
   2. USB network adapter

## Setup Procedures

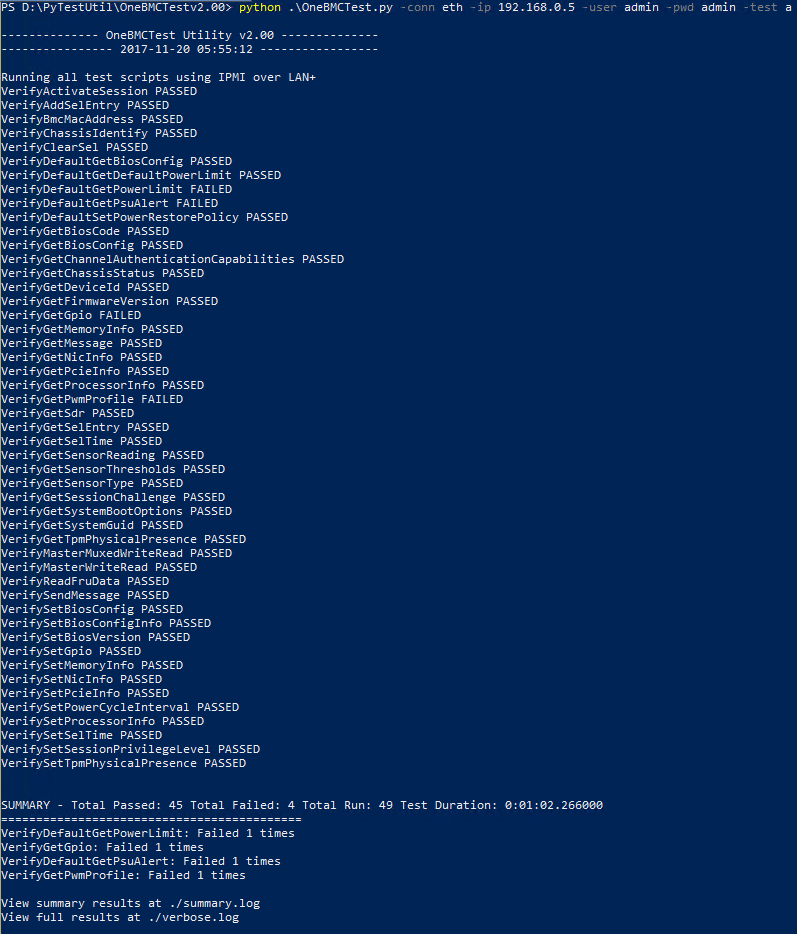
1. Place tool package in directory in operating system of either the remote machine or in-band in the compute server
2. When testing via IPMI over LAN+:
   1. Use 1 Gps Ethernet switch to connect network between remote machine and compute blade with BMC (alternative: connect remote machine directly to blade BMC)
   2. Connect Ethernet cable to BMC Ethernet port and to a USB network adapter
   3. Use IPMI command “Set LAN Configuration Parameters” to configure BMC to DHCP
   4. Set dynamic IP address in BMC by resetting BMC
   5. Verify BMC dynamic IP address is set by using IPMI command “Get LAN Configuration Parameters” and by pinging from machine that will run IpmiUtil (“ping <IpAddress>”)
   6. Note the Bmc username (<BmcUser>) and BMC password (<BmcPassword>) that can be used to run IPMI over LAN+ commands on BMC

# Running the Utility

## Quick-start

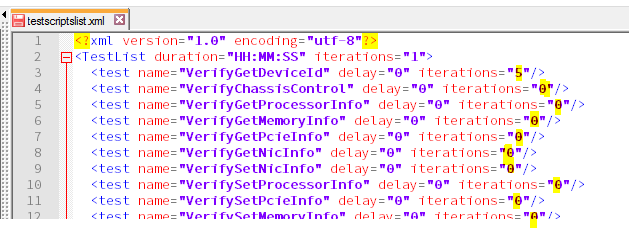
The example below will run all tests and selected tests via IPMI over LAN+.

**Run all test scripts via IPMI over LAN+**

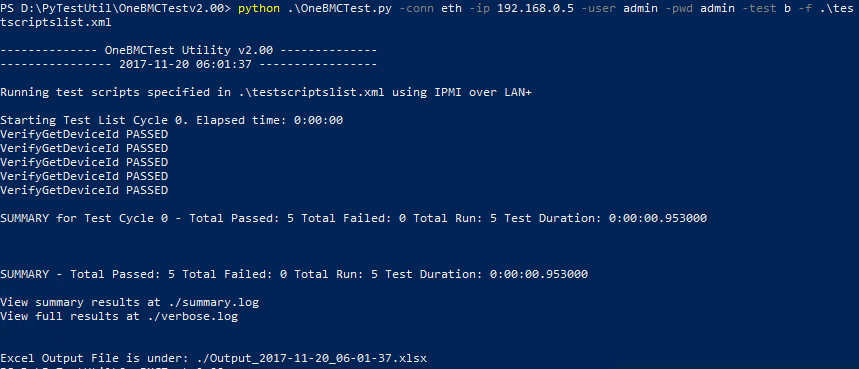
1. Open Command Prompt as Administrator and cd to OneBMCTest directory
2. Run all test scripts:
   1. python OneBMCTest.py -conn eth -ip <IpAddress> -user <BmcUser> -pwd <BmcPassword> -test a

**Run XML Batch via IPMI over LAN+**

1. Modify XML batch file to only run test script “VerifyGetDeviceId” for 5 iterations (NOTE: script being modified as example is “testscriptslist.xml”, which is included with OneBMCTest package)



1. Open Command Prompt as Administrator and cd to OneBMCTest directory
2. Run test scripts according to XML batch file via IPMI over LAN+:
   1. python OneBMCTest.py -conn eth -ip <IpAddress> -user <BmcUser> -pwd <BmcPassword> -test b -xmlfilepath “testscriptslist.xml”



## Help text

The following methods can be used to get the help text from console.

1. Run: python OneBMCTest.py
2. Run: python OneBMCTest.py –h
3. Run: python OneBMCTest.py -help

## Syntax

General syntax for using the Python Test Utility is provided below:

**Usage:** OneBMCTest.py [-h] [-platform PLATFORM] [-conn CONN] [-ip IP]

[-user USER] [-pwd PWD] [-test TEST]

[-xmlfilepath XMLFILEPATH] [-verbosename VERBOSENAME]

[-xlo EXCELOUTPUT] [-xli EXCELINPUT] [-version VERSION]

[-debug] [-showverbose] [-timestamp] [-email EMAIL]

[-sender SENDER] [-senderpwd SENDERPWD] [-server SERVER]

[-port PORT]

**Arguments:**

|  |  |
| --- | --- |
| Acceptable arguments | Argument description |
| -h  --help | show this help message and exit |
| -plt  -platform | BMC platform that is currently being tested. Default: C2010. Acceptable: C2010; J2010 |
| -conn <interfaceConnnection>  -c <interfaceConnection> | interface connection (shorthand: '-c') - options: eth (IPMI over LAN+ or J2010 REST), kcs (in-band compute server) |
| -ip <IpAddress>  -i <IpAddress> | IP address of BMC (shorthand: '-i') - required for '-conn eth' |
| -user <BmcUser>  -u <BmcUser> | User Name for connecting LAN single session or J2010 REST session (shorthand: '-u') - required for '-conn eth' |
| -pwd <BmcPassword>  -p <BmcPassword> | Password for connecting LAN single session or J2010 REST session (shorthand: ‘-p') - required for '-conn eth' |
| -test <a/b>  -t <a/b> | Tests to run (shorthand: '-t') - options: a (run all test scripts placed in <platform>TestScripts folder, b (run test scripts in <platform>TestScripts folder specified in XMLFILEPATH), <testScriptName> (run single test script with name <testScriptName>) |
| -xmlfilepath <XmlBatchFilePath>  -f <XmlBatchFilePath> | Path to XML file (shorthand: '-f') - file contains list of test scripts to run (required with '-test b') |
| -verbosename  -vbn | Name of verbose log file. File will be renamed as <verbosename>\_<timestamp>.log |
| -xlo  --excelOutput | File name for the results summary Excel file output at end of test (optional) |
| -xli  --excelInput | File name for input verbose file that can be used to create summary Excel file (optional) |
| -version <BmcVersion>  -v <BmcVersion | BMC Version string that is being tested |
| -debug  -dbg | Switch for complete IpmiUtil output in verbose output/log |
| -showverbose  -svb | Switch to print verbose logging on console output |
| -timestamp  -ts | Switch to print timestamp for every verbose log entry |
| -email <EmailAddress>  -e <EmailAddress> | Email string to send results to (required with ‘-email’ argument) |
| -sender <SenderEmailAddress>  -sn <SenderEmailAddress> | Email string for sender (required with ‘-email’ argument) |
| -senderpwd <SenderEmailPassword>  -sp <SenderEmailPassword> | Password string for sender (required with ‘-email’ argument) |
| -server <SmtpServer>  -sv <SmtpServer> | SMTP server to use to send email (required with ‘-email’ argument) |
| -port <SmtpPort>  -po <SmtpPort> | SMTP server port to use for sending email (required with ‘-email’ argument) |

# Adding Test Scripts

## Overview

All test scripts are located in either the “C2010TestScripts” folder or the “J2010TestScripts” folder. The utility scans the folder (which folder to scan is specified by using the ‘-platform’ switch) for tests to run and also for any tests specified in the XML batch file. Because of this framework, new test scripts can be added and will be automatically included when running all tests (ie ‘-test a’). In addition, the new test script can also be added to an XML batch file and will be run by OneBMCTest when using ‘-test b’.

## Creating a Test Script

**Example Test Script source code (VerifyGetDeviceId.py: verifies Success 0x00 Completion Code for Get Device Id)**

import Config

import IpmiUtil

import UtilLogger

# Prototype Setup Function

def Setup(interfaceParams):

UtilLogger.verboseLogger.info("VerifyGetDeviceId.py: running Setup fxn")

return True

# Function will test completion code for Get Device Id

def Execute(interfaceParams):

# Define Test variables

cmdPassOrFail = False

respData = None

# Define GetDeviceId variables

cmdName = 'GetDeviceId'

cmdNum = Config.cmdGetDeviceId

netFn = Config.netFnApp

# Send raw bytes via IpmiUtil

cmdPassOrFail, respData = IpmiUtil.SendRawCmd(interfaceParams, \

netFn, cmdNum, [])

# Verify response

if cmdPassOrFail:

UtilLogger.verboseLogger.info(cmdName + \

": Command passed: " + str(respData))

else:

UtilLogger.verboseLogger.error(cmdName + \

": Command failed. Completion Code: " + str(respData))

return cmdPassOrFail

# Prototype Cleanup Function

def Cleanup(interfaceParams):

UtilLogger.verboseLogger.info("VerifyGetDeviceId.py: running Cleanup fxn")

return True

The above source code serves as a template for creating a test script. The title of the test script can include any valid characters. However, it must not contain ‘\_\_’. Note: all functions written above are required in the test script as is. Thus, the following functions are required:

**Required functions in test script**

* def Setup(interfaceParams):
  + Function is used for setup of a test. This function is executed first by the utility for this test script. The function should be defined exactly as shown above.
  + Input: interfaceParams is a list of command line parameters needed to run IpmiUtil via KCS or IPMI over LAN+. If a user wants to differentiate between the current test running KCS or IPMI over LAN+, then the user can check for interfaceParams being None, which would mean that the current test script is running via KCS.
  + Output: This function needs to return either True or False. If False is returned, then the utility continues with the next test script and does not execute functions Execute and Cleanup.
  + Example usage: if we want to test Set Power Limit actually sets a value that can be retrieved by Get Power Limit, the Setup function would call Set Power Limit to set the test power limit value.
* def Execute(interfaceParams):
  + Function is used to run the actual test. This function is executed second by the utility for this test script. The function should be defined exactly as shown above.
  + Input: interfaceParams is a list of command line parameters needed to run IpmiUtil via KCS or IPMI over LAN+. If a user wants to differentiate between the current test running KCS or IPMI over LAN+, then the user can check for interfaceParams being None, which would mean that the current test script is running via KCS.
  + Output: This function needs to return either True or False. If False is returned, then the utility continues with the next test script and does not execute functions Cleanup.
  + Example usage: if we want to test Set Power Limit actually sets a value that can be retrieved by Get Power Limit, the Execute function would verify the power limit set in the Setup function by calling Get Power Limit.
* def Cleanup(interfaceParams):
  + Function is used to execute a test. This function is executed last by the utility for this test script. The function should be defined exactly as shown above.
  + Input: interfaceParams is a list of command line parameters needed to run IpmiUtil via KCS or IPMI over LAN+. If a user wants to differentiate between the current test running KCS or IPMI over LAN+, then the user can check for interfaceParams being None, which would mean that the current test script is running via KCS.
  + Output: This function needs to return either True or False.
  + Example usage: if we want to test Set Power Limit actually sets a value that can be retrieved by Get Power Limit, the Cleanup function would be used to set the power limit that was originally on the BMC by calling Set Power Limit.

**Helper functions in test script**

The utility provides the following functions that can be used for testing but are not required for running the test script:

* IpmiUtil.SendRawCmd(interfaceParams, netFn, cmd, rawBytesList):
  + This function can be used to send raw request bytes to the BMC. This function requires the IpmiUtil module (ie import IpmiUtil)
  + Input:
    - interfaceParams: list of command line parameters needed to run IpmiUtil via KCS or IPMI over LAN+.
    - netFn: hex string that represents the netFn (bits [7:2]) and lun (bits [1:0]). This value can be referenced using the Config module (ie import Config)
    - cmd: hex string that represents the IPMI command number. This value can be referenced for every IPMI command using the Config module (ie import Config)
    - rawBytesList: list of hex strings that represent the raw bytes that can be sent per IPMI request. This list only needs to contain hex strings for the IPMI request. For example, the rawBytesList for Get Processor Info for index 0x01 would be: [ ‘01’ ]. If no raw bytes need to be sent, an empty list should be used (ie []).
  + Output:
    - cmdPassOrFail: Boolean value that represents if the command returned a Success 0x00 Completion Code (True) or if it returned a completion code that is not Success (False).
    - respData: this command returns a list of hex string IPMI response bytes (if cmdPassOrFail is True) or it returns a hex string for the IPMI response Completion Code (if cmdPassOrFail is False). If the response does not return any bytes and is Success Completion Code, the respData will be 'No Response Bytes'.
* IpmiUtil.SendRawCmd2ME(interfaceParams, rawBytesList):
  + This function can be used to send raw request bytes to the ME using the Send Message command. This function requires the IpmiUtil module (ie import IpmiUtil)
  + Input:
    - interfaceParams: list of command line parameters needed to run IpmiUtil via KCS or IPMI over LAN+.
    - rawBytesList: list of hex strings that represent the raw bytes that can be sent per IPMI request. This list only needs to contain hex strings for the IPMI request. For example, the rawBytesList for get CPU and DIMM temperatures (Get CPU and Memory Temperature) would be:

[ '00', '20', 'B8', '4B', '57', '01', '00', '03', 'FF', 'FF', 'FF', 'FF', '00', '00', '00', '00' ]. If no raw bytes need to be sent, an empty list should be used (ie []).

* + Output:
    - cmdPassOrFail: Boolean value that represents if the command returned a Success 0x00 Completion Code (True) or if it returned a completion code that is not Success (False).
    - respData: this command returns a list of hex string IPMI response bytes (if cmdPassOrFail is True) or it returns a hex string for the IPMI response Completion Code (if cmdPassOrFail is False). If the response does not return any bytes and is Success Completion Code, the respData will be 'No Response Bytes'.
* IpmiUtil.RunIpmiUtil(\*args):
  + This function can be used to send command line to be executed by IpmiUtil. This function allows the user direct access to IpmiUtil, so that any command supported by IpmiUtil can be run by the user. This function requires the IpmiUtil module (ie import IpmiUtil)
  + Input:
    - \*args: variable number of arguments that will be concatenated to form the command line arguments to be run by IpmiUtil. It is recommended that \*args be a single list containing all arguments to be run by IpmiUtil. For example, in order to run “IpmiUtil.exe power -c”, set \*args to [ ‘power’, ‘-c’].
  + Output:
    - out: string that is output by IpmiUtil to stdout file stream
    - err: string that is output by IpmiUtil to stderr file stream (check err for any errors output by IpmiUtil)
* UtilLogger.verboseLogger.info(message):
  + This function can be used to log a message into ‘verbose.log’ file. This function should be used for logging information that is not an error. This function requires the UtilLogger module (ie import UtilLogger)
  + Input:
    - message: string variable that is to be logged in ‘verbose.log’ file.
  + Output: None
* UtilLogger.verboseLogger.error(message):
  + This function can be used to log an error message into ‘verbose.log’. This function requires the UtilLogger module (ie import UtilLogger)
  + Input:
    - message: string variable that is to be logged in ‘verbose.log’ file
  + Output: None

## Adding a Test Script

Once a test script is created, the user only needs to add the script to the ‘<platform>TestScripts’ folder to run the script by using ‘-test a’ (run all test scripts). However, if the user only wants to run the created test script and perhaps a few other test scripts, the user can add the script to an xml batch file and run using ‘-test b -xmlfilepath <xmlBatchFilePath>’:

<?xml version="1.0" encoding="utf-8"?>

<TestList>

<test name="NewTestScript" execute="1" iterations="1"/>

</TestList>

The “name” element value needs to be the title of the script. For the example above, the title of the script is “NewTestScript.py”.

The “execute” element is whether to run the test script (“1”) or not to run it (“0”).

The iterations element is the number of times to run the test script (“0” means the script will not run).

# Xml Batch File

## Overview

The Xml Batch file can be used to specify which test scripts in the <platform>TestScripts folder will be executed by the utility. This feature requires that the ‘-test b’ and ‘-xmlFilePath <xmlBatchFilePath>’ be used in the Command Line Interface. The Xml Batch file also includes elements to instruct the utility the number of cycles to run the sequence of specified test scripts, the length of time to run the test, the number of times to run each test script per test cycle, the amount of time to delay before running a specific test script, and whether or not to execute the test script.

## Modifying the Batch File

The following table denotes the Xml attributes and elements and how they are used:

|  |  |  |
| --- | --- | --- |
| Element | Attribute | Usage |
| TestList | duration | Describes how long to run the test. This test won’t stop until duration time has elapsed. Format is HH:MM:SS where HH is hours, MM is minutes, and SS is seconds. Example, run test for 3 hours 42 minutes, duration=”03:42:00”. To ignore this element, set as “HH:MM:SS”. |
| iterations | Integer value that describes the number of iterations to run the entire sequence of test scripts. Note: test will continue until BOTH the duration elapsed time and iterations has been satisfied. To ignore this element, set as “1” or “x”. |
| test | name | Name of the test script in <platform>TestScripts folder to be executed. This should be the name of the test scripts file without .py. For example, for test script VerifyGetDeviceId.py, name=”VerifyGetDeviceId”. |
| delay | Integer in number of seconds that determines that amount of time to delay before executing the test script. For example, delay executing VerifyGetDeviceId.py by 30 seconds: delay=”30” |
| iterations | Integer that describes the number of iterations to run the specified test script for each test cycle.  NOTE: if set to “0” or less than “0”, test script will NOT execute. |