User's Manual for Version 3.0 of the NIST DME Interface Test Suite for Facilitating Implementations of Version 1.4 of the I++ DME Interface Specification

John Horst, Thomas Kramer, Joseph Falco,
William Rippey, Frederick Proctor, and Albert Wavering
The National Institute of Standards and Technology (NIST)
U. S. Department of Commerce
Gaithersburg, Maryland, USA
Telephone: 301.975.{3430, 3518, 3455, 3417, 3425, 3461}
Email: {john.horst, thomas.kramer, joseph.falco,
william.rippey, frederick.proctor, albert.wavering}@nist.gov

October 4, 2004

"Testing a complex system is just as difficult as creating a complex system."

–Johanna Rothman, Rothman Consulting Group, Inc.

Contents

1 Version information **4**

2 Quick startup instructions: Getting started with the client-side and server-side utilities and test cases **4**
- 2.1 Downloading and unpacking the test suite . . . 4
- 2.2 Running the server-side and client-side utilities . . . 5
  - 2.2.1 Running the client-side utility . . . 5
  - 2.2.2 Running the server-side utility . . . 7
- 2.3 Using the test cases . . . 8
  - 2.3.1 Using test files of I++ DME command strings . . . 8
  - 2.3.2 Using the test artifacts . . . 8

3 Platform requirements **8**

4 How to get the most out of the entire test suite **9**
- 4.1 Using the test utilities to facilitate implementation development . . . 10
- 4.2 Using software modules to facilitate implementation development . . . 10
- 4.3 Using the test suite to improve the specification . . . 10

5 Introducing the components of the test suite **12**
- 5.1 Stand-alone components versus integrable components . . . 12
- 5.2 Who to contact for a particular test suite component . . . 14

6 Using test suite components for facilitating client-side implementations **14**
- 6.1 Using the TCP/IP socket read and write code . . . 14
- 6.2 Using the log file generation code . . . 15
- 6.3 Using the response parser code . . . 15
  - 6.3.1 Integrating the response parser in a client-side implementation . . . 15
  - 6.3.2 The response parser as stand-alone support for client-side implementations . . . 15
- 6.4 The use of client-side test utility as a pattern for client-side implementations . . . 16

7 Using test suite components for facilitating server-side implementations **16**
- 7.1 Using the TCP/IP socket read and write . . . 17
- 7.2 Using the command parser . . . 17
  - 7.2.1 The command parser as stand-alone support for server-side implementations . . . 17
- 7.3 The use of the server-side test utility as a pattern for server-side implementations . . . 18
- 7.4 Using the world model component . . . 20
- 7.5 Using the command context checker . . . 20
  - 7.5.1 The command context checker as stand-alone support for server-side implementations . . . 21
- 7.6 Trajectory generator . . . 21
- 7.7 Command executor (CMM simulator) . . . 22
- 7.8 Command and response C++ classes . . . 22

8 String formats and test file formats for commands and responses **22**
- 8.1 Format of command strings . . . 22
- 8.2 Format of command string test files . . . 24
- 8.3 Format of response strings . . . 26
- 8.4 Format of response string test files . . . 30

9 The test artifacts **31**
- 9.1 The simple test artifact . . . 31
- 9.2 The DCX test artifact . . . 33

10 Using the test suite for conformance testing **34**

11 Finding references to specific I++ DME commands in the I++ DME specification version 1.4 34

List of Figures

1 Directory structure for the NIST test suite version 3.0 5
2 The client-side utility front end 6
3 The server-side utility front end 7
4 I++ DME Interface Specification Testing and Implementation Systems 13
5 The simple test artifact 32
6 The DCX test artifact 33

List of Tables

1 A list of changes to the version 2.1 test suite now realized in version 3.0: Most of the work involved upgrading the version 2.1 test suite to be compatible with version 1.4 of the I++ DME specification. 4
2 A list of all the command string test files by type of test 9
3 A list of all the test suite components along with a brief description of what each does and where it is located . . . 10
4 A list of the test suite components and the individuals responsible for them 14
5 Page number references to I++ DME commands given in the I++ DME specification version 1.4 34

Table 1: A list of changes to the version 2.1 test suite now realized in version 3.0: Most of the work involved upgrading the version 2.1 test suite to be compatible with version 1.4 of the I++ DME specification.

Modification/improvement	**NIST Implementer**
Revised the command and response classes to bring them into compliance with version 1.4 of the I++ DME spec	Tom Kramer
Rewrote the command and response parsers to be compatible with version 1.4 of the I++ DME specification	Tom Kramer
Built a larger set of test cases in which each test case has both command and response files	Tom Kramer
Created a UNICODE version of the response parser (in response to a request from LK)	Tom Kramer
Updated world model and context checker to be compatible with version 1.4 of the I++ DME specification	Tom Kramer
Completed a converter that will transform coordinate systems so that points in each test case are within the measurement range of the particular CMM under test	Tom Kramer
Updated the server utility to incorporate all the above changes and improvements	Joe Falco
Updated the client utility to incorporate all the above changes and improvements	Joe Falco
Integrated the standalone response parser into the client utility	Joe Falco
Fixed bug where, in single step mode, the client was not waiting for ack before sending next command	Joe Falco
Fixed buffering problem in client when it receives several responses on a read socket cycle. This is especially the case with EnumProp, EnumAllProp, OnScanReport based responses.	Joe Falco
The NIST server now detects the case when a client is not waiting for an ack before sending another command and returns an error of type Protocol error - 0008.	Joe Falco

1 Version information

This user's manual describes version 3.0 of the NIST Dimensional Measuring Equipment (DME) interface test suite intended to support implementations of version 1.4 of the I++ DME specification, with the following caveat: the command GetProp() has been augmented to allow the method Tool.Radius() as an additional argument, even though Tool.Radius() is not defined in version 1.4 or any version of the I++ DME specification published at or before the date of this users manual. This slight change is in response to the expressed request of all known implementers of the I++ DME specification version 1.4, that the test suite allow tool radius information to be defined on the interface. The I++ DME specification writers team is planning an analogous change to the next version of the I++ DME specification that will address the need for tool radius information.

Version 1.4 (as well as many other versions) of the I++ DME specification are located at the following web site.

`http://www.isd.mel.nist.gov/projects/metrology_interoperability/specs/index.html`

2 Quick startup instructions: Getting started with the client-side and server-side utilities and test cases

The following instructions are for those developers who want to immediately start using the NIST test utilities and test cases. This includes those who are writing their own code from scratch and do not intend to use any of the code provided by NIST in their implementations (although we strongly encourage the use of the NIST code to save yourselves development time and reduce interoperability errors).

2.1 Downloading and unpacking the test suite

A zip file, containing all the source code, executables, and additional files pertaining to the test suite, including this user's manual (NISTI++DMEtestSuite3.0UsersManual.pdf), can be downloaded from the following link:

`http://www.isd.mel.nist.gov/projects/metrology_interoperability/resources.html`

The zip file, NISTtestSuiteVer3.0ForIppVer1.4.zip, contains a directory called NISTI++DMEtestSuite3.0. The high level directory structure of the file when it is unzipped is shown in figure 1.

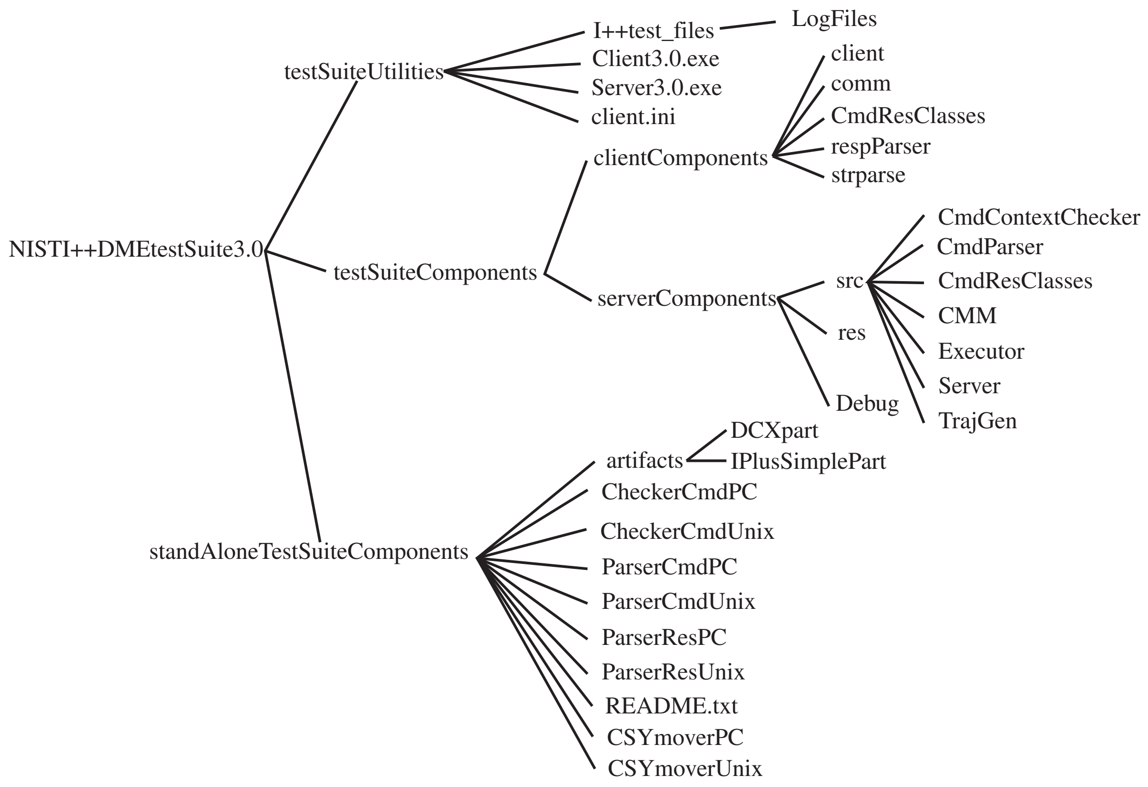


Figure 1: Directory structure for the NIST test suite version 3.0

2.2 Running the server-side and client-side utilities

The server-side and client-side utilities are located here:

```
NISTI++DMEtestSuite3.0\testSuiteUtilities\Client3.0.exe
NISTI++DMEtestSuite3.0\testSuiteUtilities\Server3.0.exe
```

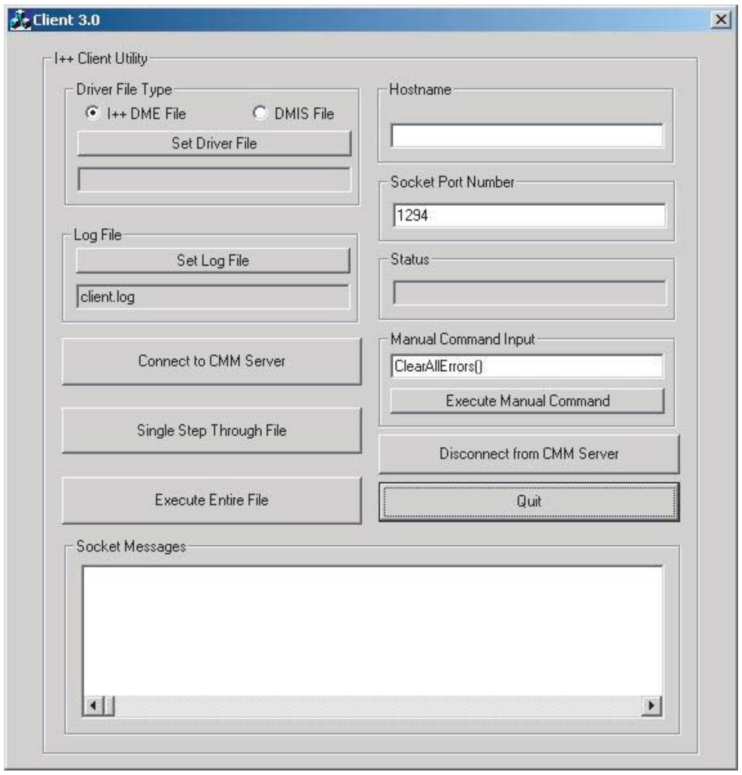
These executable files, client3.0.exe and server3.0.exe, are for testing server-side and client-side implementations of the I++ specification, respectively. For information on how to run client3.0.exe, see section 2.2.1. For information on how to run server3.0.exe, see section 2.2.2.

2.2.1 Running the client-side utility

figure 2 illustrates the Graphical User Interface (GUI) front end of the client-side utility. To start up the client-side utility, navigate to the following directory

```
 NISTI++DMEtestSuite3.0\testSuiteUtilities
```

and double click on the file, Client3.0.exe. Select which type of test program to run, either I++ DME or DMIS (Dimensional Measuring Interface Standard), though the DMIS mode is currently unsupported. Click the "Set Driver File" button and navigate to the test program file you wish to run. See section 2.3.1 for a description of the available test files provided in this test suite. Click the "Set Log File" button and navigate to the location where you wish to locate the logfile. Give the logfile a name and click Open. The log file is where the time-stamped data will be recorded.



Driver File Type: [✓ I++ DME File] DMIS File

Socket Port Number: 1294

Log File: client.log

Manual Command Input: ClearAllErrors()

Figure 2: The client-side utility front end

Within the "Hostname" box specify the hostname or IP address of the I++ DME compliant server to connect to. Enter the Hostname as either a fully qualified hostname or an IP address. One is free to manually specify the server hostname or server IP address or to keep any preexisting default hostname or IP address. The default is kept in client.ini at the same directory as Client3.0.exe. If no client.ini file existed prior to starting up the utility front end, there will obviously be no default hostname or IP address. If no hostname or IP address is manually entered prior to exiting from the utility front end, even though a client.ini file will be generated, there will be still be no default hostname or IP address in the box at the start of the next session. If a hostname or IP address was specified during a previous session, that hostname or IP address will be stored in the client.ini file at the end of that previous session, and will become the default for the next session. If the user changes the hostname or IP address during the run, the new hostname or IP address will be saved to the client.ini file and will be retrieved when the user starts the client utility again.

Specify the port number in order to create the socket. The default port number, 1294, is the one given in the I++ DME specification version 1.4. When the user clicks on the "Connect to CMM Controller" button, a non-blocking TCP/IP socket is created between the client-side utility and the CMM server on the specified port. Once the client-side utility is connected to the server, the user can either enter a command manually, single step through the program file that was selected, or run the entire file. The server implementation receives the commands, executes them, and returns the appropriate response back to the client-side utility via the socket. The client-side utility creates a time-stamped logfile of I++ DME commands sent and responses received over the socket. A status window displays the current status of the program, including the command just sent or response received as well as the existence of any error conditions. A scrolling listbox displays all messages that pass across the socket. The newest message received is at the top of the list.

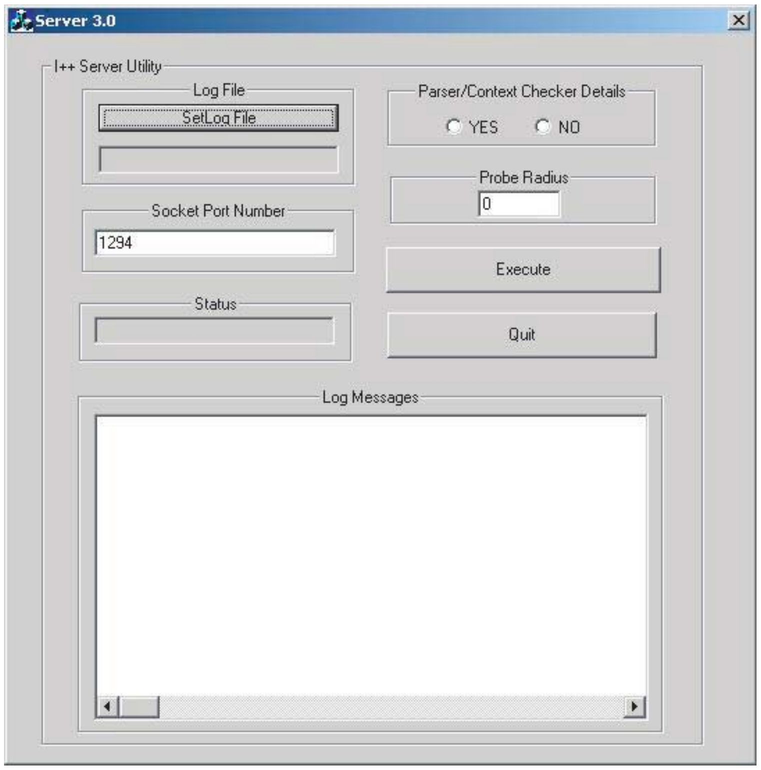


Figure 3: The server-side utility front end

2.2.2 Running the server-side utility

Since the server-side utility is set up to receive commands from the client-side, the user needs to set up and execute the server-side utility first. To execute the server-side utility, just double click the Server3.0.exe icon in the following directory.

```
NISTI++DMEtestSuite3.0\testSuiteUtilities
```

Once executing, this utility will wait indefinitely for a connection from the client. Server-side utility setup involves the following steps (refer to figure 3 for a picture of the server-side utility):

- If you wish to log test result information, click the "Set Log File" button and provide a file name. It is suggested that you set up a folder to store the log files for different test files. Command and error information are recorded in these log files. Selecting "yes" in the Parser/Context Checker Details option also specifies error logging for parsing and context checking errors. These errors are more detailed than the I++ specification compliant errors that are available as responses. They reveal more detail about why the command string parsing failed or why the command had an inappropriate context. For more detail on server-side log files, see section 7.3.
- Select the port number for TCP/IP socket communications. This number must match the one chosen on the client-side. The default port number is 1294, which is the one internationally defined for this type of connection. Therefore, it should be used whenever possible. However, other port numbers may work as long as the client-side socket uses the same numbers.
- Enter a probe radius to offset measurement values generated by the CM simulation of the probe tip.
- To execute, click the Execute button. At such point, the Status display would show "Waiting for Connection" until the client site connects, and, at such point, the server site Status display would show "Client Connected." This information continues throughout the execution of the entire test file.
- The server can be kept running through multiple sessions.

- Log files can be saved and selected between session using the "setLogFile" button. If not selected between sessions, multiple sessions will be concatenated to the currently selected log file.
- A status window displays the current status of the server including incoming commands and outgoing responses.
- A scrolling listbox displays all messages that pass across the socket. "Done" messages are placed at the top of the listbox.
- Enter a Probe Radius to receive responses from measurement commands that report values offset by a tool radius.

2.3 Using the test cases

The test cases currently consist of two items, 1) test files of I++ command strings and 2) two test artifacts. The use of test files is discussed in section 2.3.1, and the use of test artifacts is discussed in section 2.3.2. The test files and files describing the artifacts are located in the following directories:

```
NISTI++DMEtestSuite3.0\testSuiteUtilities\I++test_files
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\IPlusSimplePart
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\DCXPart
```

The test cases described above are only for testing server-side implementations. Currently, only a single test case exists for testing client-side implementations...it provides correct responses only. For the next version of the test suite, we plan to add more test cases by providing incorrect responses in order to more fully test client-side implementations.

2.3.1 Using test files of I++ DME command strings

This section describes the types and lists the names of command test files contained in the test suite version 3.0. The command test files themselves are used to test server-side implementations.

To use any one of these test files, after starting up the client-side utility, select "I++ DME" as the type of test program to run, and click the "Set Driver File" button and navigate to the particular test program file you wish to run.

The command string test files consist of lines of I++ DME command strings. Our current set of command string test files falls into five separate categories. The five categories, along with the corresponding files that belong to each category, are listed in table 2.

2.3.2 Using the test artifacts

Currently there is one test file of I++ DME command strings for the simple test artifact, Simple.prg, and one test file for the DCX test artifact, DCXpart.prg. These and other test files can be found in the following directory.

```
NISTI++DMEtestSuite3.0\testSuiteUtilities\I++test_files
```

To run the program on a real CMM with the simple test artifact being inspected, simply place any corner of the part at the machine coordinate system origin, place one side in line with the positive x-axis and the other in line with the positive y-axis in that same coordinate system.

Either the simple or the DCX artifact can be gotten from John Horst of NIST. Contact him using the byline information found in this user's manual. For CAD drawings of, and DMIS programs for, these parts also contact John Horst. For further information concerning CAD drawing, DMIS programs, and related information, see section 9. CAD files and DMIS program files for the test artifacts are located in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\IPlusSimplePart
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\DCXPart
```

3 Platform requirements

All code in this test suite is intended to be developed and run on a PC running Microsoft Visual C++ version 6.0 Professional. The server-side and client-side utilities, as well as the stand-alone component executables (see section 5.1), created in Visual C++ should run under all Windows-based operating systems (at least on Windows 95 and all subsequent versions). Certain software components in the test suite, namely, the graphical user interface (GUI) front end and the Transmission Control Protocol/Internet

Table 2: A list of all the command string test files by type of test

Type of test file	Test file names
Test files that each contain one or more instances of one type of command (plus instances of any other commands required to establish a context or to make the file usable without error in the client utility)	AbortE.prg; AbortETest_WhileMoving.prg; AlignPart.prg; AlignTool.prg; CenterPart.prg; ChangeTool.prg; ClearAllErrors.prg; DisableUser.prg; EnableUser.prg; EndSession.prg; EnumAllProp.prg; EnumProp.prg; EnumTools.prg; Error.prg; FindTool.prg; Get.prg; GetChangeToolAction.prg; GetCoordSystem.prg; GetCsyTransformation.prg; GetDMEVersion.prg; GetErrorInfo.prg; GetErrStatusE.prg; GetMachineClass.prg; GetProp.prg; GetPropE.prg; GetXtdErrStatus.prg; GoTo.prg; GoTo_Test.prg; Home.prg; Home_Test.prg; IsHomed.prg; IsUserEnabled.prg; LockAxis.prg; OnMoveReportE.prg; OnPtMeasReport.prg; OnScanReport.prg; PtMeas.prg; ReQualify.prg; ScanInCylEndIsPlane.prg; ScanInCylEndIsSphere.prg; ScanInPlaneEndIsCyl.prg; ScanInPlaneEndIsPlane.prg; ScanInPlaneEndIsSphere.prg; ScanOnCircle.prg; ScanOnCircleHint.prg; ScanOnLine.prg; ScanOnLineHint.prg; ScanUnknownHint.prg; SetCoordSystem.prg; SetCsyTransformation.prg; SetProp.prg; SetTool.prg; StartSession.prg; StopAllDaemons.prg; StopDaemon.prg; TiltCenterPart.prg; TiltPart.prg
Test files that contain some syntactical errors in command strings	numbersErrorCmd.prg, parserCmdErrors.prg
Test files that contain some logical or semantic errors in the command strings	checkerCmdErrors.prg
Test files that contain multiple correct command strings	allCmdOK.prg; allResOK.prg; numbersCmdOK.prg
Test files that contain commands for use on the simple test artifact	Simple.prg
Test files that contain commands for use on the DCX test artifact	DCXpart.prg

Protocol (TCP/IP) socket read/write component, are heavily dependent on the Visual C++ platform constraint. Others such as the parser and the context checker are not. The socket facility is Winsock version 1.1.

The stand-alone components may also be compiled and run on Unix or Linux systems, using the code in subdirectories whose names end in "Unix."

4 How to get the most out of the entire test suite

The test suite currently facilitates both client and server side implementations and it can be used in the following three ways:

1. By employing utilities that can be run against an implementation, to validate correct performance, help speed up debug efforts, and eventually test if implementations are compliant to the specification[1]
2. By making available all test suite components which can be integrated into anyone's implementation, which will
   (a) shorten implementation development times
   (b) tend to standardize and simplify implementations, further ensuring successful system interoperability in the end
3. By providing a benchmark interpretation of the specification that will be used to clarify and disambiguate the specification itself

[1]Interoperability testing is also currently in progress (an interoperability test is a test between two implementations and is usually done after development of the implementation using compliance tests). A public demonstration of interoperability between several implementations of the I++ DME specification version 1.4 is planned for the International Manufacturing Technology Show 2004. Much has been learned in the months preceding this demonstration about ambiguities in the specification and misinterpretations in the implementations. Several interoperability tests involving client and server in physically separate locations have been performed. Such remote testing is possible since the I++ DME spec defines a non-real time interface. Informal test procedures have been generated for this effort (consisting mostly of how to accomplish the remote tests, *e.g.*, via modem or over the Internet).

We now describe these three ways of using the test suite in more detail.

4.1 Using the test utilities to facilitate implementation development

This type of use is described in section 2.2.

4.2 Using software modules to facilitate implementation development

The intent of this test suite is not only to provide utilities that will ultimately test implementations for compliance, but also to provide code modules, such as command and response classes and command and response string parsers that can be used by implementers of the I++ DME specification. We believe that the use of this code can greatly facilitate implementation development (on both client and server sides of the DME interface). Even more so, we anticipate that the use of common code like that provided in the NIST I++ DME test suite, will greatly speed up the testing of implementations, allow the specification to grow into a quality, sufficiently functional, and unambiguous specification, and finally will allow the specification to move into a standards phase much more quickly.

For example, all developers of server-side implementations need a command-string parsing engine of some sort in their code. However, if all implementers agree to use the same command-string parsing engine (parsing being a highly modularizable function), *all* can make improvements to it and when testing is complete, the parsing code will be of high quality. It still would not be required by the specification, but simply helpful for developing efficient implementations. The same arguably goes for several other parts of the NIST test suite, including modules for socket read/write, command context checking, command classes, response classes, log file generation, and response context checking. These and other modules should not be required as standard by the specification, but if employed as common code in *all* implementations, a higher quality specification and ultimately a higher level of interoperability will be achieved.

Further detail on which modules are appropriate for integration into an I++ DME implementation is given in table 3, under the column entitled "Suggested integration of component information into client or server implementation source code." Detail specific to which modules are appropriate for integration into a *client-side* implementations can be found in section 6. Detail specific to which modules are appropriate for integration into a *server-side* implementations can be found in section 7.

4.3 Using the test suite to improve the specification

An unambiguous, tightly-defined, and fully functional specification is essential to ultimately ensure interoperability between various implementations. I++ DME implementations are proliferating and the test suite is being used. This has revealed weaknesses within the specification and the specification writers have been correcting these weaknesses as they become known during the development and testing phase before the specification is released as an international standard. This feedback between implementers, NIST, and the I++ DME specification writers has been copious and is documented as a Microsoft Excel spreadsheet file at the following web site, http://www.isd.mel.nist.gov/projects/metrology_interoperability/documents.html. The file is entitled, I++ specification writers responses to implementers comments.

Table 3: A list of all the test suite components along with a brief description of what each does and where it is located

Test suite component	**Description**	**Suggested integration of component information into client or server implementation source code**	**Directory location**
Client-side utility	Integrates the response parser, TCP/IP socket read and write, the response class instances, the client-side GUI front end and the client-side log file generator into a single executable utility	Useful as a model for component integration for a client-side implementation; otherwise, the client-side utility is for use in "conformance" testing	NISTI++DMEtestSuite3.0\ testSuiteComponents\ clientComponents
GUI front end for client-side test utility	Provides a front end to the I++ DME implementer that is easy to comprehend and use and also contains the "glue" code that integrates all client side components	None; Only useful for client-side test utility.	NISTI++DMEtestSuite3.0\ testSuiteComponents\ clientComponents

TCP/IP socket read and write	Writes command strings and reads response strings from the specified TCP/IP channel	Useful for integration into either client-side or server-side implementations, however this component is not written as a separable module. Nonetheless, it can easily be extracted from either the source code for the client or server utilities. Modularization is planned for a later version.	NISTI++DMEtestSuite3.0\ testSuiteComponents\ clientComponents or NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents
Response parser	As a complement to the command parser, it receives response strings as input and outputs either an error message or an instance of the appropriate response class stuffed with the values extracted from the response string	Useful for integration into client-side implementation source code, and for use with server-side implementations. Clients clearly need to parse response strings, and server implementations may want to use the standalone response parser to test response string generation code without having to connect to the NIST client utility over a socket.	The response parser (as a stand-alone component) can be found in NISTI++DMEtestSuite3.0\ standAloneTestSuiteComponents\ ParserResPC, and this same response parser code is integrated into the client-side utility at NISTI++DMEtestSuite3.0\ testSuiteComponents\ clientComponents
Client-side log file generator	The log file records time-stamped versions of all commands sent and responses received	Useful for integration into client-side implementations. The use of log files during development is essential.	Currently integrated into the client-side utility. Modularization is planned for a later version. However, it can be easily extracted from client.
Test command files	These test files consist of strings of I++ DME compliant commands. Currently consist of the following subsets of test files: 1) test files intended to test only one command, 2) test files intended to test multiple commands, 3) test files with syntax errors, 4) test files with semantic errors, and 5) test files for real execution on the "simple" artifact	None; Only useful in when used with client-side test utility in "conformance" tests	NISTI++DMEtestSuite3.0\ testSuiteUtilities\ I++test_files
Test artifacts	Two artifacts: 1) An easy to machine and cost-effective part consisting of a rectangular block and a symmetrically placed cylindrical bore hole in the middle and top of the part (see section 9.1), and 2) a part with a more complex set of features (see section 9.2)	None; only useful for interoperability testing and "conformance" testing.	NISTI++DMEtestSuite3.0\ standAloneTestSuiteComponents\ artifacts\IPlusSimplePart and NISTI++DMEtestSuite3.0\ standAloneTestSuiteComponents\ artifacts\DCXpart
Command parser	Receives command strings as input and outputs either an error message or an instance of the appropriate command class stuffed with the values extracted from the response string	Useful for integration into server-side implementations, and also for productive use with client-side implementations: Servers clearly need to parse command strings, and client implementations may want to use the standalone command parser to test command string generation code without having to connect to the NIST server utility over a socket.	The command parser (as a stand-alone component) can be found in NISTI++DMEtestSuite3.0\ standAloneTestSuiteComponents\ ParserCmdPC, and this same command parser code is integrated into the server-side utility at NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\ src\ CmdParser

Command context checker	Looks at the current command in light of the commands previously executed. If the command is illegal in context, an error is sent. If the command is legal in context, the command is sent to the executor for execution	Useful for integration into server-side implementation code	NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\src\ CmdContextChecker
Command executor (CMM simulator)	Executes commands and generates responses according to the test cases selected in the server-side GUI	Useful for integration into server-side implementation code	NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\src\Executor
Trajectory Generator	Provides a rudimentary trajectory generator for course CMM simulation	None	NISTI++DMEtestSuite3.0\ testSuiteComponents \serverComponents\src\TrajGen
Server-side GUI front end	Provides a front end to the implementation developer that is easy to comprehend and use	None; only useful for server-side test utility	NISTI++DMEtestSuite3.0\ testSuiteComponents \serverComponents\src\Server
Command and response C++ classes	Defines all the data structures and methods necessary for building instances of command and response, including the ability to generated lists (queues) of commands and responses	Useful for integration into both server-side and client-side implementation code; Must be used in order to use command executor, command parser, response parser, or command context checker.	NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\src\CmdResClasses
Server-side utility	Integrates the command parser, the context checker, the command and response class instances, the server-side GUI front end, the command executor, and the socket read/write facility into a single executable utility	Primarily for use in "conformance" testing, but also useful as a model for component integration and execution for a server-side implementation	NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\src\Server
World model	Contains data and methods for storing and maintaining information about the state of the CMM and its environment	May be useful for integration into server-side implementation code	NISTI++DMEtestSuite3.0\ testSuiteComponents\ serverComponents\src\CMM

5 Introducing the components of the test suite

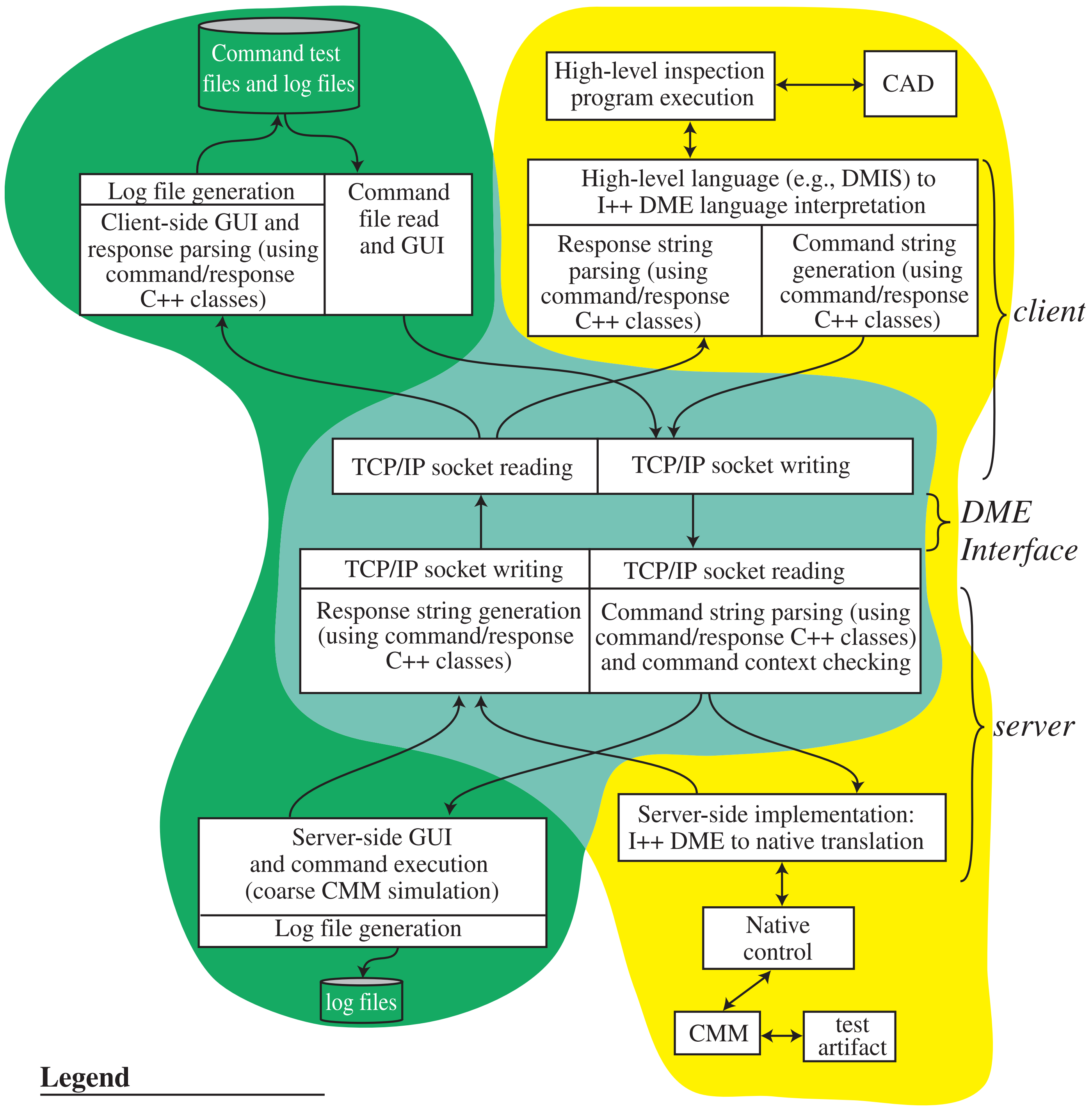
Client-side and server-side test utilities are the primary components of this test suite. They are being used to test the conformance of any implementation to the I++ DME interface spec version 1.40. However, NIST has also made a substantial effort to keep the contents of the test utility code as modular as possible in order to provide modular source code that, if used, will greatly facilitate implementation code development that is compliant with the spec and that will eventually be interoperable.

Table 3 lists all the names of the test suite components (including the test utilities), gives a brief description of the operation of each component, describes how each component can be used to facilitate high quality implementation code, and tells where in the test suite directory structure the component source code can be located.

The operation of many of these components is illustrated in figure 4. In this figure we see the components of the test suite that can be used in actual implementation code and those that normally will not. We want to encourage implementers to particularly consider the use of the command and response classes, which encapsulate so much interpretive knowledge about the specification that it will be helpful in assuring interoperability between those implementations that use the same set of command and response classes. The command and response classes also contain code to form command and response strings from the command and response data structures.

5.1 Stand-alone components versus integrable components

All the components listed in table 3 are located in NISTI++DMEtestSuite3.0\testSuiteComponents\. This directory is intended to be used by 1) developers that want to bring up the entire client-side or server-side project (or workspace) within the Visual C++ environment and immediately begin editing, compiling, and linking or 2) developers that want to include some or all of these


Command test files and log files
Log file generation
Client-side GUI and response parsing (using command/response C++ classes)
Command file read and GUI
High-level inspection program execution
CAD
High-level language (e.g., DMIS) to I++ DME language interpretation
Response string parsing (using command/response C++ classes)
Command string generation (using command/response C++ classes)
client
TCP/IP socket reading
TCP/IP socket writing
DME Interface
TCP/IP socket writing
TCP/IP socket reading
Response string generation (using command/response C++ classes)
Command string parsing (using command/response C++ classes) and command context checking
server
Server-side implementation: I++ DME to native translation
Server-side GUI and command execution (coarse CMM simulation)
Log file generation
log files
Native control
CMM
test artifact

Legend

🟩 NIST test suite for I++ DME interface specification (not used in implementations)

🟨 Actual client and server implementations (not used in test suite)

🟦 Components shared by actual implementation and test suite systems

Figure 4: I++ DME Interface Specification Testing and Implementation Systems

components in their implementations. The source code files in NISTI++DMEtestSuite3.0\testSuiteComponents\ are compiled and linked into either the client-side or server-side utility. Therefore, they are integrable versions of the components, in the sense that they cannot be compiled and executed alone, without some minor editing.

Several of the components have stand-alone versions with main() functions. These components are located in NISTI++DME testSuite3.0\standAloneTestSuiteComponents\. The stand-alone code will be helpful 1) to instruct an implementer how the component needs to be called within the implementer's code and 2) in certain cases, to test the validity of command and response strings files.

5.2 Who to contact for a particular test suite component

In order to best use the test suite components, it's good to know who to contact when questions arise. Table 4 identifies the individuals responsible for each component's development and maintenance (all of these individuals are NIST staff members and co-authors of this user's manual).

We will now describe how to fully utilize these components for developing quality implementations in an expeditious manner.

Table 4: A list of the test suite components and the individuals responsible for them

Test Suite Components	**Individuals Responsible for Component Development and Maintenance**
Client-side GUI front end	Joe Falco
Client-side utility	Joe Falco
TCP/IP socket read and write	Joe Falco
Command parser	Tom Kramer
Response parser	Tom Kramer
Client-side log file generator	Joe Falco
Test command files	Tom Kramer
Test artifacts	Joe Falco and John Horst
Command context checker	Tom Kramer
Command executor (CMM simulator)	Joe Falco
Trajectory Generator	Joe Falco
Server-side GUI front end	Joe Falco
Command and response C++ classes	Joe Falco and Tom Kramer
Server-side utility	Joe Falco

6 Using test suite components for facilitating client-side implementations

In section 4.2 we introduced the concept of using the components provided in this test suite to facilitate interoperable implementations.

The test suite version 3.0 consists of modular code as much as possible, so that, if implementers use this modular code for their implementations (of the version 1.4 specification), it will speed up code development. In table 3, we summarized how each test suite component might be used to facilitate code development of client-side or server-side implementations of the I++ DME specification for version 1.40. We will pursue this concept more fully in this section for the development of client-side implementations.

The next subsections will introduce various components and suggest how they might be used in a client-side implementation.

6.1 Using the TCP/IP socket read and write code

The code for reading from and writing to the TCP/IP socket is common for both client and server side components. Since we are restricting implementation platforms to be a PC with a Windows operating system, the socket read/write code component can be easily integrated into implementation code, saving the implementer much time and effort. So, the socket read/write component in the test suite is one of those components that is used in the executing utilities (both server and client side) and can be used in implementation code, as is illustrated in figure 4.

6.2 Using the log file generation code

The log file generation code is modularizable, but not currently defined as a separate module. However, if any implementers wish to extract the log file generation code themselves, they are able to. Simply access the file, ServerDlg.cpp, located in

```
\NISTI++DMEtestSuite3.0\testSuiteComponents\serverComponents\src\Server
```

6.3 Using the response parser code

A standalone response parser, that is constructed in a similar manner to the command parser, has been developed, tested, and integrated into the client-side test utility. Therefore, the standalone response parser can readily be integrated into client-side implementation source code. Clients clearly need to parse response strings, and use of NIST's response parser should greatly facilitate client-side implementation development.

Additionally, server-side implementations may want to use the standalone response parser to test response string generation code without having to connect to the NIST client utility over a socket.

All files relating to the PC and UNIX versions, respectively, of the standalone response parser can be found in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserResPC\
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserResUnix\
```

and all directories listed in this section are subdirectories of ParserResPC only, since the UNIX version has only minor modifications. The PC software compiles under Visual C++ (version 6.0 Professional) consistent with our platform constraints (see section 3).

Now we describe these two uses of the response parser in more detail.

6.3.1 Integrating the response parser in a client-side implementation

To use the response parser in your own client-side implementation, the setInput() method must first be called to copy an input string into the parser's inputArray.

The caller should then call parseResponse(). If there is no error in the input string, parseResponse makes an instance of a Response from the input string and returns a pointer to that instance. If there is an error, parseResponse returns a NULL pointer.

After calling parseResponse, the caller should call getParserErr() to see if there was an error. If the returned error code is not OK, the caller can call getErrorMessage(code) to get an error message that describes the error. If the error code is OK, the caller can start processing the Response instance. The caller may call parseTag() before calling parseResponse(), but that is not required.

6.3.2 The response parser as stand-alone support for client-side implementations

An executable for the stand-alone response parser, which should run on any Windows operating system, is:

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserResPC\bin\parserRes.exe
```

Sample response files are in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserResPC\test_files
```

To use the executable on a PC, open a command window and change directories to:

```
  NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserResPC
```

Then give the following command, bin\parserRes.exe

The executable will start up and ask whether you want stepping on or off. After you enter y or n, it will ask for a file name. Enter a file name such as, test_files\allResOK.res.

If you turned stepping on, you have to keep hitting the <enter> key to step through the file. If not, the entire file will be read and processed.

After the file has been completely read, the executable will prompt you to enter either another file name or q to quit.

Descriptions of the format of response strings and response string files are in sections 8.3 and 8.4. The test_files subdirectory contains the test files:

- allCmdOK.res

- allResOK.res
- errorResponses.res
- parserResErrors.res

The command files (.prg suffix) that correspond to the first two response files are also in that subdirectory. There is no .prg file for the other two response files.

For each .res file there is also a .txt file that explains the contents of the .res file.

The stand-alone response parser works as follows. The main function reads response strings from the response file, and calls the parser's parseResponse method. If parsing succeeds, parseResponse makes an instance of a Response and returns a pointer to it. The main function then prints the response from the Response instance (not by simply reprinting the text of the response string) followed by \\ on a separate line. If parsing fails, parseResponse returns a NULL pointer. The main function then prints the text of the response string followed by the error message caused by the response string.

To test your own response strings, put them in a file in the required format, and run the parser as described above.

The Unix version of the response parser may be run the same way as described above, except use slashes (not backslashes) in path names, and work in the directory:

```
NISTI++DMEtestSuite3.0/standAloneTestSuiteComponents/ParserResUnix
```

6.4 The use of client-side test utility as a pattern for client-side implementations

How to use the client-side utility has been described in section 2.2.1. The client-side utility is much less complex than the server-side utility in both content and execution. However, the manner in which the response parser, log file generator, and the TCP/IP write and read module are integrated into the client-side utility will be useful as a pattern for developing a client-side implementation.

There are two execution threads in the client utility. Each thread represents a cyclically executing piece of code. One thread reads command strings from a file and writes them to a socket. The other thread reads response strings from the socket. The contents of the write thread are as follows:

- Read a command string from a file. If the command strings in the file do not include tag numbers, the client will assign them.
- If an ack response for the previously sent command has been received, write the command string to the socket.
- Log the command string placed on the socket.

The client can be directed to automatically traverse an entire text file of command strings, or can be placed in single step mode so a user can step though the file. At any time, the user also has the option to manually enter a command string and request to write it to the socket. The contents of the read thread are as follows:

- The client reads the socket for responses
- For each response on the socket the client:
  - Logs the response
  - Checks the validity of the response using the response parser
  - If the response is invalid, the response parser error is logged
  - If the response is an ack, a flag is set to read the next command during the execution of the write thread.

7 Using test suite components for facilitating server-side implementations

In section 4.2 and in table 3, we introduced the concept of using the components provided in this test suite to facilitate interoperable implementations.

The test suite version 3.0 consists of modular code as much as possible, so that, if implementers use this modular code for their implementations (of the version 1.4 specification), it will speed up code development. In table 3, we summarized how each test suite component might be used to facilitate code development of client-side or server-side implementations of the I++ DME specification for version 1.40. We will pursue this concept more fully in this section for the development of server-side implementations.

The next subsections will introduce various components and suggest how they might be used in a server-side implementation.

7.1 Using the TCP/IP socket read and write

This code is virtually identical to the socket read/write component in the client-side components as described in section 6.1.

7.2 Using the command parser

The command string parser is a separately defined software component with precisely specified interfaces. A integrable version of the parser has been integrated with the overall server-side utility (section 2.2.2) and it also exists in a stand-alone version (with a main()). Developers are encouraged to utilize the integrable and stand-alone versions of the command parser in the following ways,

- In the integrable version as part of the server-side utility for facilitating client-side implementations
- In the integrable version for integration into an implementer's server-side implementation
- As a stand-alone version for I++ command files to test if input commands are parsable.

Descriptions of the format of command strings is in section 8.1 and command string test files is in section 8.2 under the heading, Command test file format.

All files relating to the integrable and stand-alone versions of the parser, respectively, described in this section can be found in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserCmdPC
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserCmdUnix
```

7.2.1 The command parser as stand-alone support for server-side implementations

An executable for the stand-alone command parser, which should run on any Windows operating system, is:

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserCmdPC\bin\parserCmd.exe
```

Sample command files are in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserCmdPC\test_files
```

To use the executable on a PC, open a command window and change directories to:

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\ParserCmdPC
```

Then give the following command, bin\parserCmd.exe.

The executable will start up and ask whether you want stepping on or off. After you enter y or n, it will ask for a file name. Enter a file name such as, test_files\allCmdOK.prg

If you turned stepping on, you have to keep hitting the <enter> key to step through the file. If not, the entire file will be read and processed.

After the file has been completely read, the executable will prompt you to enter either another file name or q to quit.

Descriptions of the format of command strings and command string files are in sections 8.1 and 8.2. The test_files subdirectory contains the test files:

- allCmdOK.prg
- allResOK.prg
- numbersCmdOK.prg
- parserCmdErrors.prg

The response files (.res suffix) that correspond to the first three command files are also in that subdirectory. There is no .res file for parserCmdErrors.prg.

For each .prg file there is also a .txt file that explains the contents of the .prg file.

The stand-alone command parser works as follows. The main function reads command strings from the command file, and calls the parser's parseCommand method. If parsing succeeds, parseCommand makes an instance of a Command and returns a pointer to it. The main function then prints the command from the Command instance (not by simply reprinting the text of the command string) followed by \\ on a separate line. If parsing fails, parseCommand returns a NULL pointer. The main function then prints (1) the text of the command string followed by (2) the error message caused by the command string, followed by (3) the I++ error number of the error.

To test your own command strings, put them in a file in the required format, and run the parser as described above.

The Unix version of the command parser executable (bin/parserCmd) may be run the same way as described above, except use slashes (not backslashes) in path names, and work in the directory:

```
NISTI++DMEtestSuite3.0/standAloneTestSuiteComponents/ParserCmdUnix
```

A Sun Solaris[2] executable is in bin/parserCmd. If your machine is not running Sun Solaris, you can compile an executable for your machine by getting into the directory listed above, changing the first two lines of the Makefile to refer to your compiler and then giving the command "make bin/parserCmd".

7.3 The use of the server-side test utility as a pattern for server-side implementations

The use of the server-side utility has been summarized in section 2.2.2. However, more detail needs to be given to gain a basic understanding of this utility, particularly if one is using the server-side utility as a pattern for developing a server-side implementation.

The server-side utility is the component consisting of mostly "glue" code that integrates all the other server-side components into a single executable. The server-side utility executes the socket read of commands, socket write of responses, command string parsing, command context checking, world model, and command execution, *i.e.*, coarse CMM simulation (see table 3 for more detail). The operation of these components is illustrated in figure 4. The server-side utility maintains "world model" data and methods files (world.h and world.cpp) that contain data about the virtual CMM and about the state of the system (*e.g.*, the context of commands) and methods needed to update and maintain that data in world.cpp.

There are two execution threads in the server-side utility. Each thread represents a cyclically executing piece of code. The contents of the first thread are as follows:

1. Check the socket for data
2. If there is data in the socket, the data is checked to ensure that the client is only sending one commend string transaction at a time to the server.
3. If one commend string is sent, the proposed command string is then parsed and checked for a correctly formed tag and to ensure that the tag number is not currently in use.
4. If more than one command string is sent, an error response is
5.
6. If there is data in the socket, the data is parsed and checked for a correctly formed tag and also to ensure that the tag number is not currently in use.
7. If the tag is legal, the command string is put in either the fast or slow queue.
8. If the tag is illegal, an error response is generated (and logged, if that option is selected) and a response string is put in the response queue.
9. The response queue is checked to see if it has responses to send back to the client; if it does, then the response queue is emptied, sending the response strings to the client.
10. The code waits until the end of the sampling period for this thread, then returns to step number 1.

The contents of the second thread are as follows:

1. Check the fast queue[3] for a command string.

[2] Certain commercial companies and their equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply any judgment by the National Institute of Standards and Technology concerning the companies or their products, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

[3] see the I++ specification for definitions of slow and fast queues. Briefly, any command in the fast queue is executed prior to execution of commands in the slow queue.

2. If there is no command string in the fast queue, it checks the slow queue for a command string.
3. If a command string was found from either queue and a multicycle command (that is, any of the seven scan commands, any of the three form tester command, AlignPart(), AlignTool(), Home(), GoTo(), or PtMeas()) is not currently executing, the parser is called to check the command string for validity (*i.e.*, the parser checks for syntax errors and parameter errors).
4. If a multicycle command is currently executing and there is a command string on the fast queue, the parser is called to check the command string for validity.
5. If a multicycle command is currently executing and there is a command string on the slow queue, but none on the fast queue, the queues are left alone and the command string is not parsed.
6. If the command string is not valid, an error response is generated and a response string is put in the response queue.
7. If the command string is valid, the parser returns a command object (*i.e.*, an instance of the appropriate command class).
8. The command object is sent to the context checker.
9. If the command has bad context, an error response is generated and a response string is put in the response queue.
10. If the context of the command is good, the command object is given to the executor. If, in addition, the command object is AbortE(), clear all queues.
11. If the command is a GoTo, Home, PtMeas, or a scan command, the command object is given to the trajectory generator.
12. The executor is checked for responses. During this check, the trajectory generator is first checked for activity. If active with a command, it is time stepped and any responses are passed to the executor. All current executor responses are then sent to the response queue.
13. The code waits until the end of the sampling period for this thread, then returns to step number 1.

Having two threads of execution allows us to set a unique cycle time (*i.e.*, sampling rate) for the execution of each thread. Typically we want to have the first thread at a higher sampling rate and the sampling rate of the second thread be less than the rate of the first, since there is no need to check the command queues, if you have not read the socket for a new command since you last checked the command queues for the presence of commands. Currently, the first thread has a sampling period of 20 ms and the second thread, 50 ms.

If the logging option is selected in the GUI (see section 2.2.2), the log file will include information on commands received, responses sent, and errors detected. The log file has the following general format:

- Timestamp, received command tag number, command string
- Timestamp, serial number of the command, acknowledgement symbol, &
- Timestamp, tag number of the command, followed by either only a command completion symbol, or any other additional information that the I++ DME specification calls for after the completion of a command
- Timestamp, tag number of the command, error report for the command
- If there is an error in the command string (detected by the command parser), the I++ error type is logged
- If "yes" is selected in the Parser/Context Checker Details option, a more detailed description of the error is also logged
- If there is an error in the command context and "yes" in the Parser/Context Checker Details option is selected, a detailed description of that error is logged

The log file may include additional execution status information users have programmed into the server utility. For example, current tool position, execution state, etc. This will be explained further in section 7.7 on the executor.

7.4 Using the world model component

In order to further modularize the server-side components, we organized the various variables (and the methods required to maintain those variables) relating to the CMM system and its environment (*i.e.*, the world model) into a separate set of source code files, world.h, world.cpp, and tools.h. These files are contained in the directory

```
NISTI++DMEtestSuite3.0\testSuiteComponents\serverComponents\src\CMM
```

The file, world.h, defines data members and methods for a world modeler that keeps track of the state of a system executing I++ DME commands. The methods in executor.cpp and serverDlg.cpp use many of the methods and data definitions in world.h and world.cpp. serverDlg.cpp is the file for the server-side GUI front end and contains much of the "glue" code for the server-side utility.

For those I++ DME specification server-side implementers who wish to integrate world.h into their own server code, we encourage them to use world.h and world.cpp as a template, *i.e.*, to use some of the constructs, replace some of the constructs, and add new constructs.

These files contain a substantial amount of code to support coordinate system transformations. However, we expect that many developers implementing I++ DME on the server-side will want to keep their proprietary version of world.h, world.cpp, and tools.h. Nonetheless, world.h, world.cpp, and tools.h are required for operation of the server-side utility.

7.5 Using the command context checker

The command context checker is a separately defined software component with precisely specified interfaces. The context checker maintains a record in the world model (world.cpp and world.h) of the previous commands. Using constraints in the specification, the context checker determines the legality of the current command, given its context.

A integrable version of the context checker has been integrated with the overall server-side utility (section 7.3) and it also exists in a stand-alone version, *i.e.*, with a main(). The main() function uses both the command parser and the checker so that it can be used with the test files of section 8.2. Developers are encouraged to utilize the integrable and stand-alone versions of the command context checker in the following ways,

- In the integrable version, as part of the server-side utility, for facilitating testing client-side implementations
- In the integrable version for integration into an implementer's server-side implementation
- In the stand-alone version for I++ command files to check that commands are legal in context.

A description of the format of command strings is in section 8.1 and a description of command string files is in section 8.2 under the heading, Command test file format.

All files relating to the integrable and stand-alone versions of the context checker described in this section can be found for the PC and for UNIX (SUN systems), respectively, in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\CheckerCmdPC
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\CheckerCmdUnix
```

The UNIX version of the context checker, checkerCmd.cc, will compile under GNU g++ and is located among the stand-alone components. The PC version of the context checker, checkerCmd.cpp, will compile in Visual C++ and is located among both the integrable and stand-alone components.

The source code defines functions in the checker class and defines a main function outside the checker class. Functions named checkXXX (where XXX is a command name) are used to check each type of command in context. The arguments to each command are assumed to have passed the checks performed by the parser. The documentation for each of these functions gives the rules that it is enforcing and gives one or more references to pages of version 1.4 of the I++ DME specification. If this checker is used without the checks performed by the parser having been made previously, the checker may crash or give wrong results.

Several semantic checks are made in the context checker that could be made in the parser (because context is not required). For example, the check that a direction vector is not (0, 0, 0). These are identified in the source code as semantic checks.

The checker class, defined in checkerCmd.h, should make it self-evident how the checker component is intended to be used. The documentation of main in the source code describes how main uses the checker class.

7.5.1 The command context checker as stand-alone support for server-side implementations

An executable for the stand-alone command context checker, which should run on any Windows operating system, is:

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\CheckerCmdPC\bin\checkerCmd.exe
```

Sample command files are in

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\CheckerCmdPC\test_files
```

To use the executable on a PC, open a command window and change directories to:

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\CheckerCmdPC
```

Then give the following command, bin\checkerCmd.exe.

The executable will start up and ask whether you want stepping on or off. After you enter y or n, it will ask for a file name. Enter a a file name such as, test_files\allCmdOK.prg

If you turned stepping on, you have to keep hitting the <enter> key to step through the file. If not, the entire file will be read and processed.

After the file has been completely read, the executable will prompt you to enter either another file name or q to quit.

Descriptions of the format of command strings and command string files are in sections 8.1 and 8.2. The test_files subdirectory contains the test files:

- allCmdOK.prg
- checkerCmdErrors.prg

For checkerCmdErrors.prg there are two .txt files; one shows what the command context checker prints, and the other shows what the command parser prints.

The stand-alone command context checker works as follows. The main function reads command strings from the command file, and calls the parser's parseCommand method.

If parsing fails, parseCommand returns a NULL pointer. The main function then prints 1) the text of the command string followed by 2) the error message caused by the command string, followed by 3) the I++ error number of the error.

If parsing succeeds, parseCommand makes an instance of a Command which is then checked to see if it is legal in context by checkCommand.

If the command is legal in context, the main function then 1) prints the command from the Command instance (not by simply reprinting the text of the command string) followed by \\ on a separate line, and 2) updates the world to simulate having executed the command.

If the command is not legal in context, the main function then prints 1) the text of the command string followed by 2) the checker error message caused by the command string, followed by 3) the I++ error number of the error.

To test your own command strings in context, put them in a file in the required format, and run the checker as described above. If you write your own command file, it is a good idea to run the checker every time you add a few commands to the file because any error will cause the context to require that a ClearAllErrors command be given before any other command can execute without error. Alternatively you can just put a lot of extra ClearAllErrors commands in the file, even though you expect them not to be needed.

The Unix version of the command context checker executable (bin\checkerCmd) may be run the same way as described above, except use slashes (not backslashes) in path names, and work in the directory:

```
NISTI++DMEtestSuite3.0/standAloneTestSuiteComponents/CheckerCmdUnix
```

A Sun Solaris executable is in bin\checkerCmd. If your machine is not running Sun Solaris, you can compile an executable for your machine by getting into the directory listed above, changing the first two lines of the Makefile to refer to your compiler and then giving the command "make bin\checkerCmd".

7.6 Trajectory generator

The trajectory generator (TrajGen) is a separate software component with separate source code and clearly defined interfaces. The trajectory generator is integrated into the command executor. The command executor passes GoTo, Home, PtMeas, or scan commands to the trajectory generator. The trajectory generator simulates time stepped moves and generates response data for OnScanReprort, OnPtMeasReport, and OnMoveReport commands.

7.7 Command executor (CMM simulator)

The command executor is a separate software component with separate source code and clearly defined interfaces. The command executor is integrated into the server-side utility. The command executor acts in the following roles:

- As a coarse CMM simulator, including the operation of the TrajGen software component
- As a locus for generating server-side test cases (generally erroneous responses of various types)
- As a separate component, so that the executor, of all the components linking into the server-side utility, is the only one that would need to be replaced by proprietary code in order to develop a server-side implementation. This should be clear from figure 4.

7.8 Command and response C++ classes

A common set of command and response classes does not seem to be required by the I++ DME specification, however their use by all implementers is highly encouraged, in order to achieve a high level of system interoperability in the end. Using common command and status classes in implementations will reduce development and debug time and will streamline the testing and analysis process.

NIST has defined a set of I++ DME specification compliant command and response classes. Accompanying these classes are C++ files defining various methods for each of these classes. In order to simplify the class structure, the actual command and response classes are derived classes from command and response base classes, respectively. The primary function of these classes is to provide a common set of data structures for passing data and generating command and response strings. The role of the command and response classes is illustrated in figure 4. Also defined are classes for handling data types and errors defined within the specification. The data classes contain the necessary logic for formatting the data according to the specification when a command or response string is being generated.

8 String formats and test file formats for commands and responses

This section describes the details of the unique formats for command strings, command string test files, response strings, and response string test files.

8.1 Format of command strings

This document covers the syntax of individual command strings. For a command string to be legal in a session, it is necessary but not sufficient that its syntax conform to the requirements given here. Constraints on the situations in which a command string may be given exist, but are not included here.

Any string purported to be a command string but failing to conform to the requirements given here should cause an error of some type to be issued by the server.

1. What is an I++ Command String?

   An I++ command string is a string of characters intended to be put into a character string to be transmitted by a client through a communications system to an I++ DME Interface server. The command string represents an I++ DME Interface command. See section 8.2 for details on command files and character strings.

2. Use of ASCII

   All references to ASCII characters in this command string spec are given using decimal (not octal or hexadecimal) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

3. Character Set

   All bytes in a command string are to be interpreted as 8-bit ASCII characters. The 8th bit must always be zero.

   Only characters in the range from <32> (space) to <126> ( ) may be used, except that the character pair carriage return <13> and line feed <10> is used as a command string terminator.

   The characters <13> and <10> may not occur anywhere else in a command string other than at the end, and in the order <13><10>.

   Upper case letters and lower case letters are regarded as different letters. In other words, command strings are case sensitive.

4. Command String Length

   The maximum number of characters in a command string, including the <CR><LF>, must not exceed 65536.

5. Numbers

   Numbers are defined in the following hierarchy.

   ```
   number
       integer
           unsigned_four_digit_integer
       decimal_point_number
           no_exponent_number
           exponent_number
   ```

   A digit is one of the characters 0 to 9 (<48> to <57>).

   An unsigned_four_digit_integer consists of exactly four digits. Example: 0287

   An integer consists of an optional plus <43> or minus <45> sign followed by one to sixteen digits. Example: +287741

   Note that every unsigned_four_digit_integer is also an integer.

   A no_exponent_number consists of an optional plus or minus sign followed by either: a. a decimal point <46> followed by one to sixteen digits, or b. one or more digits followed by a decimal point followed by zero or more digits so that the total number of digits before and after the decimal point is not more than sixteen. Examples: (a) -.3090 (b) 5.31

   An exponent_number consists of a no_exponent_number as just described followed by an E <69> followed by by an optional plus or minus sign followed by one, two, or three digits. Example: -2.8843E02

   A decimal_point_number is either an exponent_number or a no_exponent_number.

   A number is either an integer or a decimal_point_number.

   The values of all types of numbers are to be interpreted in the normal way as base ten numbers.

6. Strings

   A string consists of a double quote <34>, followed by zero to many of the characters allowed by section 3 in the I++ DME spec (excluding <34>, <13>, and <10>) followed by a double quote.

7. Tags

   There are two types of tags: CommandTag and EventTag

   A CommandTag is formed by putting any digit except zero before an unsigned_four_digit_integer. The command tag 00000 may not be used.

   An EventTag is formed by putting an upper case E <69> before an unsigned_four_digit_integer. The EventTag E0000 may not be used in a command string.

   Examples of CommandTags:

   ```
   04711 // tag is OK
   00020 // tag is OK
   20 // error; only 2 characters
   00000 // error; explicitly disallowed.
   ```

   Examples of EventTags:

   ```
   E4711 // tag is OK
   E0333 // tag is OK
   e0333 // error; illegal first character, must use upper case E
   E0000 // error; explicitly disallowed
   E20 // error; only 3 characters
   A4711 // error; first character not E
   ```

   The first 5 characters of a command string represent a tag.

8. Command String Syntax

   A command string consists of the following, in order

   (a) a tag
   (b) a space
   (c) a method name
   (d) a left parenthesis <40> optionally preceded and followed by any number of spaces <32>
   (e) an optional argument_list
   (f) a right parenthesis <41> optionally preceded (but NOT followed) by any number of spaces <32>
   (g) a carriage return line feed pair <13><10>

   An argument_list consists of one or more arguments separated by commas. A comma must not be placed after the last argument. Each comma may optionally be preceded and followed by any number of spaces <32>.

   Allowed arguments are as described in Section 6.1.42.2 of the spec. A wide range of arguments, that are not easily described, are allowed.

9. Examples

   The following are example of potentially legal command strings. They may become illegal in context.

   00001 StartSession()<13><10>

   01095 GoTo(X(-0.75), Y(1.0), Z(-.00))<13><10>

   07003 GetProp(FoundTool.PtMeasPar.Speed())<13><10>

8.2 Format of command string test files

1. What is an I++ DME Interface Command File?

   An I++ DME Interface Command File is a file containing character strings to be stuffed into messages and sent to an I++ DME Interface server. Each character string represents a legal or illegal command message. The command file also contains character string separator sequences and end of file sequences that are not part of the character strings. In this spec "character string" will be used to refer to the characters that go into the message.

2. Suffix

   Command files are identified by a ".prg" suffix.

3. Communications

   It is assumed here that sockets are being used for communications. When sockets are used, the length of the character string being transmitted is given in the communication, and no terminator (such as a NULL) is used in the character string.

   To be suitable for use with some other communications method, this file format may need to be modified.

4. Use of ASCII

   All references to ASCII characters in this file spec are given using decimal (not octal or hexadecimal) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

5. How the Command File is Divided

   The first character string of a command file starts with the first character in the file and ends on the last character before the first occurrence of two backslashes followed by a carriage return followed by a line feed (*i.e.*, <92><92><13><10>). The second character string starts with the next character in the file after that and ends on the first character before next occurrence of <92><92><13><10>, and so on. The <92><92><13><10> sequence is a separator and is not part of any character string.

   The backslashes are used so that character strings representing illegal commands with <13><10> inside can be written in the file and used for testing.

   To end the file, after the <92><92><13><10> following the last character string, there should be the sequence <58><13><10> <58><13><10>. This has the appearance of two lines each containing only a colon.

Using ASCII for non-printing characters, here is an example (written on two lines) of an entire command file with two character strings in it, each representing a legal command:

```
00001 StartSession()<13><10>\\<13><10>
00002 EndSession()<13><10>\\<13><10><58><13><10><58><13><10>
```

The first character string is: 00001 StartSession()<13><10> The second character string is: 00002 EndSession()<13><10>

When this file is viewed in most file viewers, it has the following appearance:

```
00001 StartSession()
\\
00002 EndSession()
\\
:
:
```

6. Legal Character Strings

   A legal character string consists of a legal command string, as defined in 8.1, and nothing else.

   A legal command string followed by any other characters before the separator forms an illegal character string.

7. Multiple Sessions

   Files of legal commands must always begin with a StartSession command. Files that represent complete sessions must always end with an EndSession command. Multiple sessions may be included in a single file.

8. Comments

   Command files contain no comments. This is to keep parsing easy. It is intended that a .txt (text) file with the same base name accompany each .prg file. The .txt file should explain the .prg file.

   Reading of command files is expected to stop when the two colons ending the file are encountered. Thus, for most command file readers, anything after the colons is effectively a comment. Users who choose to do so can put comments there.

9. Examples

   In these examples, the character string separators are represented as \\ alone on a line, and it is assumed each line ends with <13><10>.

   (a) All Legal File

   In this file, all the character strings represent legal commands given in a legal order.

   ```
   StartSession()
   \\
   Home()
   \\
   GoTo(X(3), Y(-2.0))
   \\
   EndSession()
   \\
   :
   :
   ```

   (b) All Illegal Character Strings File

   In this file, all the characters strings are illegal. This is helpful in order to insure that server-side implementations can handle different types of illegally formed command strings.

   ```
   StartSession()
   hi mom
   \\
   home()
   \\
   GoTo(X3, Y-2)
   ```

```
\\
:
:
```

The first character string has "hi mom" after the first <13><10> and before the end of the string. The second character string spells the command "home" starting with a lower case h (must be upper case). The third character string fails to put the X and Y values in parentheses.

8.3 Format of response strings

Now we look at the syntax of individual response strings.

General issues relating to response strings: Any string purported to be a response string but failing to conform to the requirements given here should cause an error in the client.

1. What is an I++ response string?

   An I++ response string is a string of characters intended to be put into a character string to be transmitted by an I++ DME Interface server through a communications system to a client. The response string represents an I++ DME Interface response.

2. Use of ASCII (American Standard Code for Information Interchange)

   The use of ASCII is as described in section 8.1. In this file, an ASCII character may be denoted by a decimal integer in angle brackets. For example, <32> is the ASCII space character.

3. Character set

   The use of characters is as described in section 8.1.

4. Response string length

   The number of characters in a response string, including the <13><10> at the end, must not exceed 65536. This is the same as the length requirements for commands.

5. Numbers

   The format of numbers is as described in section 8.1.

6. Strings

   The format of strings is as described in section 8.1.

7. Tags

   The format of tags is as described in section 8.1. The EventTag E0000 may be used in a response string to indicate errors that are not the result of executing a command or cannot be identified with a specific command.

   If a response can be identified with a command, the tag in the response must be the same as the tag in the command.

8. Commas

   Whenever a comma <44> is used, it may optionally be preceded or followed by any number of spaces <32>. Wherever the use of a comma is described below, it is implicit that the optional space may be used.

9. Response string syntax

   A response string consists of the following, in order

   (a) a tag.
   (b) a space.
   (c) a single character that is a response type indicator. This must be one of &<38>, %<37>, #<35>, or !<33>.
   (d) zero to many continuation characters as described below.
   (e) a carriage return line feed pair <13><10>.

   If the response type indicator is & or %, there are zero continuation characters.

   If the response type indicator is ! or #, the first continuation character is always a space.

Error response: If the response type indicator is !, this indicates an error. The additional continuation characters consist of the following, in order.

1. "Error" (without the quotes).
2. a left parenthesis <40> optionally preceded and followed by any number of spaces <32>.
3. a single character that must be one of 1 2 3 9. This character must be the severity character as given in section 8.2 of the spec for the error number described in 5 (below).
4. a comma.
5. a number consisting of four digits (with no sign and no decimal point) that is one of the error numbers given in section 8.2 of the spec.
6. a comma.
7. a string.
8. a comma.
9. a string. This string must be the text as given in section 8.2 of the spec for the error number described in item 5 (above).
10. a right parenthesis <41> optionally preceded (but NOT followed) by any number of spaces <32>.

Data response: If the response type indicator is #, this indicates data is being returned. The additional continuation characters are one of following. In each of these data responses, a left parenthesis may always be preceded or followed by any number of spaces <32>, a right parenthesis may always be preceded (but NOT followed) by any number of spaces, and a comma may always be preceded or followed by any number of spaces.

1. Align Data

   This is used in response to an AlignTool command or an AlignPart command. It has the following, in order.

   (a) A left parenthesis.
   (b) either two or five numbers, each followed by a comma.
   (c) a number.
   (d) a right parenthesis.

   Example 1a: 00001 # (1.000000,0.000000,0.000000)
   Example 1b: 00001 # (1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000)

2. Property type data

   This is used in response to either an EnumProp or an EnumAllProp command. It has the following, in order:

   (a) a string giving the name of a property. This must be one of: "Tool", "FoundTool", "GoToPar", "PtMeasPar", "Speed", "Accel", "Approach", "Retract", "Search", "Min", "Max", "Act", or "Def".
   (b) a comma.
   (c) a string giving the name of a data type. This must be one of:"Number", "Property", "String". If item a) above is "Tool", "FoundTool", "GoToPar", or "PtMeasPar", this must be "Property". If item a above is "Speed", "Accel", "Approach", and "Retract", or "Search", , this must be "Number" or "Property". If item a) above is "Min", "Max", "Act", or "Def", this must be "Number".

   Example 2a: 00001 # "Speed", "Number"
   Example 2b: 00001 # "Tool","Property"

3. Coordinate System Type Data

   This is used only in response to a GetCoordSystem command. It has the following, in order:

(a) "CoordSystem" (without the quotes).
(b) a left parenthesis.
(c) one of: MachineCsy, MoveableMachineCsy, MultipleArmCsy, PartCsy.
(d) a right parenthesis.

Example 3a: CoordSystem(MachineCsy)

4. Coordinate System Transformation Data

This is used only in response to a GetCsyTransformation command. It has the following, in order:

(a) "GetCsyTransformation" (without the quotes).
(b) a left parenthesis.
(c) one of: PartCsy, JogDisplayCsy, JogMoveCsy, SensorCsy, MultipleArmCsy.
(d) six numbers, each preceded by a comma.
(e) a right parenthesis.

Example 4a: 00001 # GetCsyTransformation(PartCsy,1, -2.00,3, 30.0, 45.0,20.0)

5. String Data

This is used only in response to a GetErrorInfo or EnumTools command. It consists of a single string. A single GetErrorInfo or EnumTools command may elicit several responses.

Example 5a: 00001 # "no clue"

6. Error Status Data

This is used only in response to a GetErrStatusE command. It consists of the following, in order:

(a) "ErrStatus" (without the quotes).
(b) a left parenthesis.
(c) a single character that is either 0 or 1.
(d) a right parenthesis.

Example 6a: E0001 # ErrStatus(0)

7. Machine Class Data

This is used only in response to a GetMachineClass command. It consists of the following, in order:

(a) "GetMachineClass" (without the quotes).
(b) a left parenthesis.
(c) the string "CartCMM" (with the quotes) or the string "CartCMMWithRotaryTable" (with the quotes).
(d) a right parenthesis.

Example 7a: 00001 # GetMachineClass("CartCMM")

8. Is Homed Data

This is used response to an IsHomed command or a GetXtdErrStatus command. It consists of the following, in order:

(a) "IsHomed" (without the quotes).
(b) a left parenthesis.
(c) a single character that is either 0 or 1.
(d) a right parenthesis.

Example 8a: 00001 # IsHomed(1)

9. Is User Enabled Data

   This is used in response to an IsUserEnabled or a GetXtdErrStatus command. It consists of the following, in order:

   (a) "IsUserEnabled" (without the quotes).
   (b) a left parenthesis.
   (c) a single character that is either 0 or 1.
   (d) a right parenthesis.

   Example 9a: 00001 # IsUserEnabled(1)

10. Property data

    This is used in response to a GetProp, GetPropE, or SetProp command. It consists of the following in order:

    (a) the keyword "Tool" or "FoundTool" (without the quotes).
    (b) a dot <46>.
    (c) the keyword "PtMeasPar" or "GoToPar" (without the quotes).
    (d) a dot.
    (e) one of the following keywords (without the quotes): "MaxSpeed", "Speed", "MinSpeed", "MaxAccel", "Accel", or "MinAccel". If the preceding keyword is "PtMeasPar", this may also be "MinApproach", "Approach", "MaxApproach", "MinRetract", "Retract", "MaxRetract", "MinSearch", "Search", or "MaxSearch" (without the quotes).
    (f) a left parenthesis.
    (g) a number.
    (h) a right parenthesis.

    Example 10a: 00001 # Tool.PtMeasPar.MinAccel(2.000000)

11. Position data

    This is used in response to a Get, PtMeas, or OnMoveReportE command. It consists of one to seven parts separated by commas. Each part consists of the following, in order:

    (a) one of "R", "X", "Y", "Z", "Tool.A", "Tool.B", or "Tool.C" (without the quotes).
    (b) a left parenthesis.
    (c) a number.
    (d) a right parenthesis.

    Each of R, X, Y, Z, Tool.A, Tool.B, and Tool.C may appear at most once, but they may appear in any order.

    Example 11a: 00001 # Y(2.000000), Z(3.000000)
    Example 11b: 00001 # X(1.2)
    Example 11c: 00001 # X(1.2),Z(3.0), Y(2.0)

12. Scan data

    This is used in response to a ScanOnCircle, ScanOnLine, ScanInPlaneEndIsCyl, ScanInPlaneEndIsPlane, ScanInPlaneEndIsSphere, ScanInCylEndIsPlane, or ScanInCylEndIsSphere command. It consists of numbers separated by commas.

    Example 12a: 00001 # 0.0, 1.0, 2.0, 0.5, 1.5, 2.0

8.4 Format of response string test files

1. What is an I++ DME interface response file?

   An I++ DME Interface Response File is a file containing character strings to be stuffed into messages and sent to an I++ DME Interface client. Each character string represents a legal or illegal response message. The response file also contains character string separator sequences and end of file sequences that are not part of the character strings. In this spec "character string" will be used to refer to the characters that go into the message.

2. Suffix

   Response files are identified by a ".res" suffix.

3. Communications

   It is assumed here that sockets are being used for communications. When sockets are used, the length of the character string being transmitted is given in the communication, and no terminator (such as a NULL) is used in the character string.

   To be suitable for use with some other communications method, this file format may need to be modified.

4. Use of ASCII

   All references to ASCII characters in this file spec are given using decimal (not octal or hex) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

5. How the Response File is Divided

   The first character string of a response file starts with the first character in the file and ends on the last character before the first occurrence of two backslashes followed by a carriage return followed by a line feed (*i.e.*, <92><92><13><10>). The second character string starts with the next character in the file after that and ends on the first character before next occurrence of <92><92><13><10>, and so on. The <92><92><13><10> sequence is a separator and is not part of any character string.

   The backslashes are used so that character strings representing illegal responses with <13><10> inside can be written in the file and used for testing.

   To end the file, after the <92><92><13><10> following the last character string, there should be the sequence <58><13><10> <58><13><10>. This has the appearance of two lines each containing only a colon.

   Using ASCII for non-printing characters, here is an example (written on two lines) of an entire response file with two character strings in it, each representing a legal response:

   ```
   00001 &<13><10>\\<13><10>
   00001 %<13><10>\\<13><10><58><13><10><58><13><10>
   ```

   The first character string is: 00001 &<13><10>
   The second character string is: 00001 %<13><10>

   When this file is viewed in most file viewers, it has the following appearance:

   ```
   00001 &
   \\
   00001 %
   \\
   :
   :
   ```

6. Legal Character Strings

   A legal character string consists of a legal response string, as defined in 8.3, and nothing else.

   A legal response string followed by any other characters before the separator forms an illegal character string.

7. Comments

   Response files contain no comments. This is to keep parsing easy. It is intended that a .txt (text) file with the same base name accompany each .res file. The .txt file should explain the .res file and the corresponding .prg file, if there is one.

   Reading of response files is expected to stop when the two colons ending the file are encountered. Thus, for most response file readers, anything after the colons is effectively a comment. Users who choose to do so can put comments there.

8. Examples

   In these examples, the character string separators are represented as \ \ alone on a line, and it is assumed each line ends with <13><10>.

   (a) All Legal File

   In this file, all the character strings represent legal responses.

```
00001 &
\ \
00012 # IsHomed(1)
\ \
00015 # Y(2.000000), Z(3.000000)
\ \
00015 %
\ \
:
:
```

   (b) All Illegal Character Strings File

   In this file, all the characters strings are illegal.

```
00001 &
oops
\ \
00003 # (1.0, 2.0)
\ \
00004 # Isuserenabled(1)
\ \
:
:
```

   The first character string has "oops" after the first <13><10> and before the end of the string.

   The second character string does not correspond to any valid response format.

   The third character string has two lower case letters where they should be upper case in IsUserEnabled.

9 The test artifacts

The ultimate goal of this effort is to create 1) a specification for the equipment (DME) interface of a CMM and 2) a test suite that together will help discriminate between implementations that are fully functional and interoperable and those that are are not.

In the following two sections, we describe two artifacts currently in use in support of interoperability tests (of I++ DME implementations) being conducted worldwide, often remotely. The role of these artifacts (and the inspection process plans supporting them) is to expose defects of functionality and interoperability early on in the specification development process.

9.1 The simple test artifact

The "simple" artifact is so-called, because it seemed good to have an initial artifact that possessed the following characteristics:

- easy and cost-effective to manufacture
- has an uncomplicated set of features
- naturally exposes interoperability problems uncomplicated by metrology issues

The simple artifact coupled with a file of I++ DME compliant commands constitute a test case. These test cases form part of the entire test suite. The simple artifact can be easily and quickly manufactured by any I++ DME implementer desiring to test the validity of their implementation using the test cases by using the CAD files provided in our test suite distribution.

The simple part is represented in the following CAD formats:

- IGES
- ProEngineer
- STEP_Part21_AP203

The simple part is represented in the following CAD kernel formats:

- Parasolid
- ACIS10.0

These CAD formats are found in appropriate subdirectories of the following directory (see figure 1).

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\IPlusSimplePart\
```

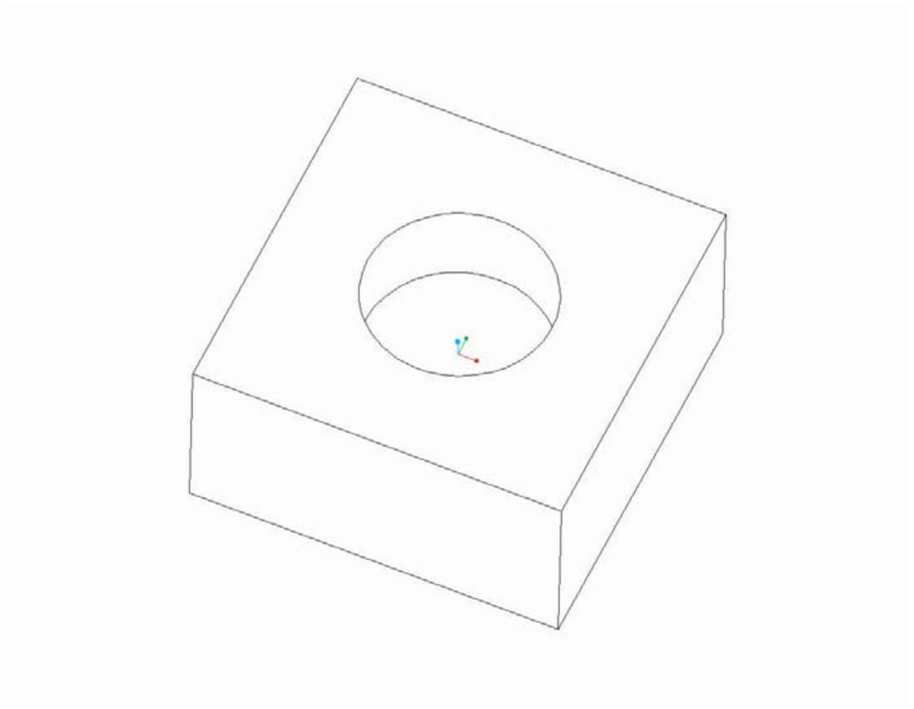


Figure 5: The simple test artifact

A DMIS inspection program, simple_in.dms, has been written for this part as well as the corresponding DMIS output file, output.dms, are located in the following directory within the test suite distribution.

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\IPlusSimplePart\DMIS
```

An I++ DME command file, Simple.prg, has also been written for this part, which is consistent with the DMIS program, simple_in.dms, and is located in the following directory.

```
NISTI++DMEtestSuite3.0\testSuiteUtilities\I++test_files
```

9.2 The DCX test artifact

The DCX artifact is a part with a more complex set of features intended to contain many of the features commonly encountered in manufacturing metrology. This part is being used at the live demonstration of I++ DME interoperability held at the International Manufacturing Technology Show (IMTS) 2004 in Chicago, USA.

The DCX artifact has the advantage (over the simple artifact) that it is appropriate for testing implementations of dimensional metrology system interface standards for *all* the key interfaces, including DME interface standards (e.g., I++ DME), CAD interface standards (e.g., STEP APs), inspection process plan interface standards (e.g., DMIS) and reporting and analysis interface standards (e.g., DML).

The DCX artifact coupled with an inspection process plan constitute a test case. These test cases form part of the entire test suite. The simple artifact can be easily and quickly manufactured by any I++ DME implementer desiring to test the validity of their implementation using the test cases by using the CAD files provided in our test suite distribution.

The DCX part is represented in the following CAD formats:

- IGES
- ProEngineer
- STEP_Part21_AP203

The DCX part is represented in the following CAD kernel formats:

- Parasolid
- ACIS10.0

These CAD formats are found in appropriate subdirectories of the following directory (see Figure 1).

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifacts\DCXPart\
```

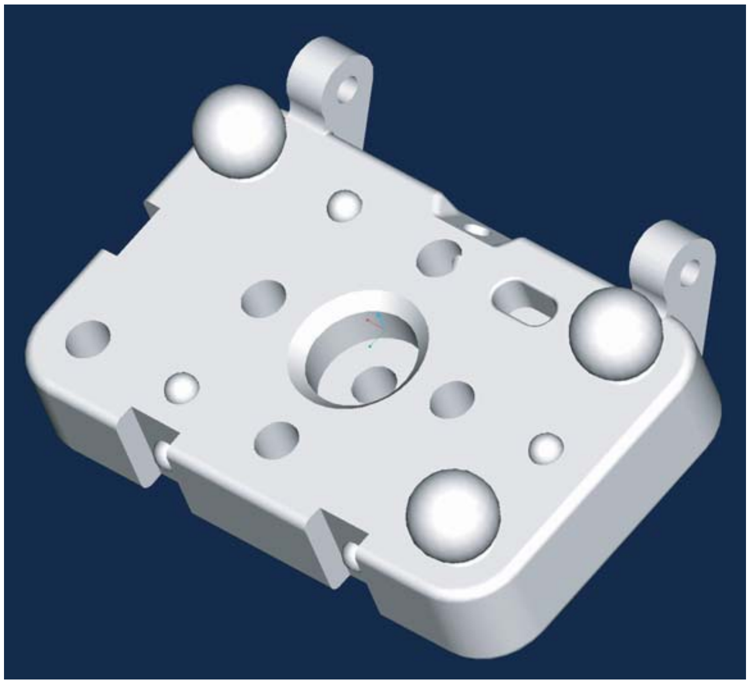


Figure 6: The DCX test artifact

A DMIS inspection program, IMTS_M_clean.dmi, has been written for this part and it is located in the following directory within the test suite distribution.

```
NISTI++DMEtestSuite3.0\standAloneTestSuiteComponents\artifactsDCXPart\DMIS
```

An I++ DME command file, DCXpart.prg, has also been written for this part, which is consistent with the DMIS program, IMTS_M_clean.dmi, and is located in the following directory.

```
NISTI++DMEtestSuite3.0\testSuiteUtilities\I++test_files
```

10 Using the test suite for conformance testing

The goal of developing a test suite is to allow a potential purchaser (the user) of inspection software (client) or a CMM (server) to be able to apply a conformance test to the client or server prior to purchase. Placing a level of performance of implementation scores on conformance tests should substantially increase the level of interface interoperability for dimensional metrology systems.

The NIST I++ DME test suite is being expanded to enable semi-formal conformance tests, *i.e.*, the components of this expanded test suite will be used to test the conformance of any single implementation of the specification.

Conformance can only be asymptotically approximated; perfect conformance and perfect interoperability can never be achieved, due to the inability of the test suite to identify every possible misinterpretation of any interface language specification. Furthermore, it is practically impossible (without a formally defined specification) to define a completely unambiguous specification. Any ambiguity is ripe for misinterpretation, and not testable.

Though we are not currently providing a conformance test *per se*, our goal is to approximate it. Release 3.0 offers a more comprehensive set of test cases that more closely approximates a conformance test than did the test cases of release 2.1. section 2.3.1 describes the use of the test cases for release 3.0. These test cases facilitate both server-side and client-side implementation development, however, the server-side test cases for client-side implementations consist of only one test case. This test case always produces correct responses, however we have place-holders for adding more server-side test cases which can create incorrect responses of various types. We plan to include these test cases in release 3.1. At that point, we will have a test suite that contains an *informal* conformance test, which can be employed by the purchasing departments of metrology system users corporations to verify compliance with the I++ DME specification and make informed purchasing choices.

The client-side test cases consist of two test artifacts and test inspection programs. The test inspection programs consist of text files of I++ DME compliant command strings and a specification for the format of that text file. Server-side test cases currently consist of a single test case always returning correct responses. Our next release is planned to have multiple server-side test cases that return a variety of incorrect response patterns and syntax. Our client-side utility also produces a log file to facilitate debugging and ultimately to use for testing compliance.

11 Finding references to specific I++ DME commands in the I++ DME specification version 1.4

Table 5 gives an index to I++ DME commands found in release 1.4. The numbers refer to page numbers. The pages listed refer to both text and examples.

The command references are prioritized and categorized in table 5 as follows. The most important references are given in parentheses, example references are given in brackets, and other references are not enclosed.

For example,

```
AbortE: 23 26 (39) [40] 85
```

means the most important text is on page 39 and an example is on page 40.

For any command, page 17 is never referenced because the print is too small. Page 25 is not referenced since everything on it appears identically elsewhere.

Table 5: Page number references to I++ DME commands given in the I++ DME specification version 1.4

I++ DME 1.4 commands	Page numbers giving locations of prioritized and categorized references to each

AbortE	9 24 26 (42) [43] 99
AlignPart	24 28 63 (95) 105
AlignTool	24 27 29 (55) 66 70 102 107
CenterPart	24 27 (96) 102
ChangeTool	24 27 29 (54) (57) (58) (78) 98 102 106
ClearAllErrors	24 26 [38] (40) (41) (42) (43) [44] (47) (75) 76 99
DisableUser	24 27 (48) (51) (53) 102
EnableUser	23 27 [37] (44) 88
EndSession	18 20 24 26 (41) (77) 99
EnumAllProp	24 26 45 (46) [74] 99
EnumProp)	24 26 29 (45) [73] 99 108 109 110 111
EnumTools	24 29 (53) (54) (56) 106
Error (response only)	20 26 30 31 33-35 [36] 37 38 [38] 39 [39] 40 43 [43] [44] (75-76) 100 101 113 These reference pages cover only the format and use of error messages. They do not cover the specific errors that can occur for each command. They do not cover ClearAllErrors, GetErrStatusE, or GetXtdErrStatus.
FindTool	24 27 29 (54) [74] 102 106
Get	24 27 [38] [44] (49) (51) 57 (61) (63) (66) (81) 102
GetChangeToolAction	24 27 29 (57-58) 102 106
GetCoordSystem	24 28 (59) 103
GetCsyTransformation	24 28 (60) (80) 103
GetDMEVersion	24 26 (46) [47] 99
GetErrorInfo	19 24 26 (43) 76 99 Page 76 does not mention the command but gives allowable error numbers.
GetErrStatusE	24 27 (50) 102
GetMachineClass	18 24 27 (50) 102
GetProp	24 26 29 (45) 66 67 [68] [69] 70 [73] [74] 77 79 99 108
GetProp(Tool.Alignment())	24 29 (70) 107
GetProp(Tool.CollisionVolume())	24 29 (67-70) [68] [69] 107
GetProp(Tool.Id())	24 29 (66) 107
GetProp(Tool.Name())	24 29 (66) 107
GetPropE	24 26 29 (45) 99 108 Also see reference pages for GetProp
GetXtdErrStatus	24 27 (50) 102
GoTo	24 27 [38] [43] (51 - 53) (58) (62) [71] [72] (78) (98) 102
GoTo (server-initiated response)	(78)
Home	24 27 [38] (48) 102
IsHomed	24 27 (48) 102 111
IsUserEnabled	24 27 (49) 102
KeyPress (server-initiated response)	(78-79)
LockAxis	24 (97-98)
OnMoveReportE	19 24 27 [39] (49 - 50) 102
OnPtMeasReport	24 27 (41) (49) (52) (53) 61 [71] [72] (78) 102
OnScanReport	24 27 (41) (81) 82 83 84 86 87 88 90 91 [93] 103
PtMeas	9 24 27 (41) (49) (52 - 53) (62) [71] [72] (78) 83 84 86 87 89 90 91 102
PtMeas (server-initiated response)	(78)
ReQualify	24 29 (64) 106 107 Called Qualify on pages 29 106 107
Scan (response)	24 27 (41) (81) 82 83 84 86 87 88 90 91 [93] 103
ScanInCylEndIsPlane	24 27 (90 - 91) 103
ScanInCylEndIsSphere	24 27 (89 - 90) 103

ScanInPlaneEndIsCyl	24 27 (88 - 89) 103
ScanInPlaneEndIsPlane	24 27 (86 - 87) 103
ScanInPlaneEndIsSphere	24 27 (85 - 86) [94] 103
ScanOnCircle	24 27 (82 - 83) [93] 103
ScanOnCircleHint	24 27 (82) [93] 103
ScanOnLine	24 27 (83 - 84) 103
ScanOnLineHint	24 27 (83) 103
ScanUnknownHint	24 27 (85) [93] 103
SetCoordSystem	24 28 58 (59) (67) [71] [72] 103
SetCsyTransformation	24 28 59 (60) 67 [71] [72] (80) 103
SetProp	24 26 29 (45) 65 [73] 77 (78) 99 108 Also see the reference pages for GetProp, since several of those imply that certain properties cannot be set directly.
SetProp (server-initiated response)	(78)
SetTool	24 27 29 (54 - 55) 102 106
StartSession	18 20 24 26 (38) (41) 49 [71] (77) 81 99
StopAllDaemons	24 26 (42) (50) 99
StopDaemon	19 (22) 24 26 [40] (42) (50) 99
TiltCenterPart	24 27 (97) 102
TiltPart	24 27 (97) 102
command arguments in general	27 28 (49) (52 - 53) (61) (62) (78) (81) 103 104
number (definition)	(33 - 34)
space (definition and use)	(33 - 34)
string (definition and use)	10 (33) 34 35 43 46 66 73 99 102 106 107 108 109 These references are only those where string is the data type of a component of a command or a response. The word string is used in other places to refer to an entire command or response.
tag	(21) (22) 26-30 (34-42) (47) (49) (50) (75) (77) 99-113