

# **User's Manual for Version 2.1 of the NIST DME Interface Test Suite for Facilitating Implementations of Version 1.30 of the I++ DME Interface Specification**

John Horst, Thomas Kramer, Joseph Falco, Keith Stouffer,  
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, 3877, 3427, 3434, 3425, 3461}

Email: {john.horst, thomas.kramer, joseph.falco, keith.stouffer,  
frederick.proctor, albert.wavering}@nist.gov

February 4, 2004

"In software development, testing should<sup>4</sup> occur:

1. At the beginning
2. In the middle
3. On those who think testing should occur only at the end"

---

*—a poster advertisement for Rational Software Corporation*

# Contents

<b>1</b>	<b>Version information</b>	<b>5</b>
<b>2</b>	<b>Quick startup instructions: Getting started with the client-side and server-side utilities and test cases</b>	<b>5</b>
2.1	Downloading and unpacking the test suite . . . . .	6
2.2	Running the server-side and client-side utilities . . . . .	6
2.3	Using the test cases . . . . .	7
<b>3</b>	<b>Platform requirements</b>	<b>7</b>
<b>4</b>	<b>How to get the most out of the entire test suite</b>	<b>7</b>
4.1	Using the executables to facilitate implementation development . . . . .	7
4.2	Using software modules to facilitate implementation development . . . . .	7
4.3	Using the test suite to improve the specification . . . . .	8
<b>5</b>	<b>Using the test suite for conformance and interoperability testing</b>	<b>8</b>
<b>6</b>	<b>Introducing the components of the test suite</b>	<b>8</b>
6.1	The two sets of components within the test suite: client-side components and server-side components . . . . .	8
6.2	Stand-alone components versus stripped-down components . . . . .	9
6.3	Who to contact for a particular test suite component . . . . .	11
<b>7</b>	<b>Using the client-side components for facilitating server-side implementations</b>	<b>11</b>
7.1	Operating the client-side GUI front end . . . . .	12
7.2	Using the TCP/IP socket read and write code . . . . .	13
7.3	Using the log file generation code . . . . .	13
7.4	Using the command string test files . . . . .	14
7.4.1	Format of command strings . . . . .	14
7.4.2	Format of command string test files . . . . .	17
7.5	Using the simple test artifact . . . . .	18
7.6	Using the response parser code . . . . .	20
7.6.1	Using the response parser as stand-alone support for server-side implementations . . . . .	20
7.6.2	Integrating the response parser in a client-side implementation . . . . .	21
7.6.3	Format of response strings . . . . .	21
7.6.4	Format of response string test files . . . . .	25
<b>8</b>	<b>Using server-side components for facilitating client-side implementations</b>	<b>26</b>
8.1	Operating the server-side GUI front end . . . . .	26
8.2	Server-side utility . . . . .	27
8.3	TCP/IP socket read and write . . . . .	29
8.4	Using the command string parser . . . . .	29
8.4.1	Using the command parser as stand-alone support for client-side implementations . . . . .	29
8.5	CMM and tools related components . . . . .	30
8.6	Command context checker . . . . .	30
8.6.1	Using the command context checker as stand-alone support for client-side implementations . . . . .	31
8.7	Trajectory generator . . . . .	32
8.8	Command executor (CMM simulator) . . . . .	32
8.9	Command and response C++ classes . . . . .	32
<b>9</b>	<b>Finding references to specific I++ DME commands in the I++ DME specification version 1.30</b>	<b>33</b>

## List of Figures

1	Directory structure for the NIST test suite version 2.1 . . . . .	6
2	I++ DME Interface Specification Testing and Implementation Systems . . . . .	10
3	The client-side GUI front end . . . . .	13
4	The simple test artifact . . . . .	19
5	The server-side GUI front end . . . . .	27

## List of Tables

1	A list of improvements to the version 2.0 test suite now realized in version 2.1 . . . . .	4
2	A list of all the client-side components along with a brief description of what each does and where it is located . . .	9
3	A list of all the server-side components along with a brief description of what each does and where it is located. . .	11
4	A list of the test suite components and the individuals responsible for them . . . . .	12
5	A list of all the command string test files by type of test . . . . .	14
6	Page number references to I++ DME commands given in the I++ DME specification version 1.30 . . . . .	34

Table 1: A list of improvements to the version 2.0 test suite now realized in version 2.1

Problem Reported by	Problem Description	Solution	NIST Implementer
Metromec (René Keller) & Wilcox Associates (Michele Penlae)	Some error messages are received without being followed by %	A bug was fixed in the NIST server code	Joe Falco
Metromec (René Keller)	GetProp using some cases of multiple parameters producing error. ( <i>i.e.</i> 00003 GetProp(Tool.GoToPar.Speed(), Tool.PtMeasPar.Retract()))	A bug was fixed in the NIST Command Parser	Tom Kramer
Zeiss (Josef Resch)	ScanOnCircle/OnScanReport - in some cases reports too many points	Fixed a problem in the NIST server's trajectory simulation	Joe Falco
Zeiss (Josef Resch)	Scanning and PtMeas measurements do not report values that are not offset by a tool radius	Added the capability to enter a probe radius in the Server GUI. The trajectory simulator uses this value when executing PtMeas and Scanning commands.	Joe Falco
Metromec (René Keller)	NIST's implementation treats E0001 and 00001 as the same tag number.	The NIST implementation was revised to recognize E0001 and 00001 as two distinct tag numbers.	Joe Falco
NIST (Joe Falco)	The NIST Server implementation is not resetting report flags for OnMoveReport, OnPtMeasReport, and OnScanReport between sessions.	Tom Kramer provided a method in the NIST implementation World class to reset all report flags to false.	Tom Kramer
NIST (Tom Kramer)	The NIST implementation world updating function does nothing when a SetTool command is executed. The SetTool command should have the same effect on the world as theChangeTool command – the current tool is set to the one named in the command.	The action of SetTool in the world update method was changed so that the active tool is changed (to the tool named in the SetTool command) when SetTool is executed.	Tom Kramer
NIST (Tom Kramer)	The NIST server should set the various maxima and minima of parameters to something realistic during initialization.	<p>The initial maxima and minima for the following parameters were changed to more realistic values during initialization of the server:</p> <p>For PtMeasPars(max/min):</p> <ul style="list-style-type: none"> <li>Approach distance (50/0.1 mm)</li> <li>Retract distance (50/0.1 mm)</li> <li>Search distance (50/0.1 mm)</li> <li>Speed (200/0.1 mm/s)</li> <li>Acceleration (1000/0.1 mm/s<sup>2</sup>)</li> </ul> <p>For GoToPars(max/min):</p> <ul style="list-style-type: none"> <li>Speed (500/0.1 mm/s)</li> <li>Accel (2000/0.1 mm/s<sup>2</sup>)</li> </ul>	Joe Falco

NIST (Tom Kramer)	During NIST server initialization, we should add at least one probe to the toolchanger so that we can write command files that perform inspection with an actual probe.	During server initialization, a probe named "Probe1" is added to the toolchanger.	Joe Falco
NIST (Tom Kramer)	GetCsyTransformation command reports back wrong information	Fixed	Joe Falco
NIST (Joe Falco)	Following an error of severity $\geq 2$ , the queue are still processed and the server keeps sending the error "Use Clear All Errors To Continue"	Server now explicitly clears the fast and slow queues upon generating an error of severity $\geq 2$	Joe Falco
NIST (Tom Kramer)	There are circumstances where the simulation runs away.	Fixed a bug in the linear move trajectory generation code	Joe Falco
NIST (Tom Kramer)	AlignTool and AlignPart are not returning data to the client	Fixed	Joe Falco
NIST (Joe Falco)	All error severities (even those of severity less than 2) were requiring ClearAllErrors to continue	Fixed a type incompatibility in the server	Joe Falco
NIST (Tom Kramer)	"Argument out of range" errors are not being reported	This error handling capability was added to the Server Side Utility world and executor code	Tom Kramer & Joe Falco
NIST (Joe Falco & Tom Kramer)	There were inconsistencies in the format for reporting the F3 variable in error responses	Example:  Invalid command: 00020 Home(23)  Error Response: 00020 ! Error(3, 0502, "00020 Home(23)", "Incorrect parameters")  Error Response (Parser/Context Details enabled): 00020 ! Error(3, 0502, "00020 Home(23) : HOME MUST HAVE NO ARGUMENTS", "Incorrect parameters")	Joe Falco

## 1 Version information

This user's manual describes version 2.1 of the NIST Dimensional Measuring Equipment (DME) interface test suite intended to support implementations of version 1.30 of the I++ DME specification. This and 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](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.

## 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, can be downloaded from the following link:

[http://www.isd.mel.nist.gov/projects/metrology\\_interoperability/resources.html](http://www.isd.mel.nist.gov/projects/metrology_interoperability/resources.html)

A pdf file, NISTI++DMETestSuite2.1UsersManual.pdf, which can also be downloaded from the same link, contains this user's manual.

The zip file, NISTtestSuiteVer2.1ForIppVer1.30.zip, contains a directory called NISTI++DMETestSuite2.1. The high level directory structure of the file when it is unzipped is shown in Figure 1.

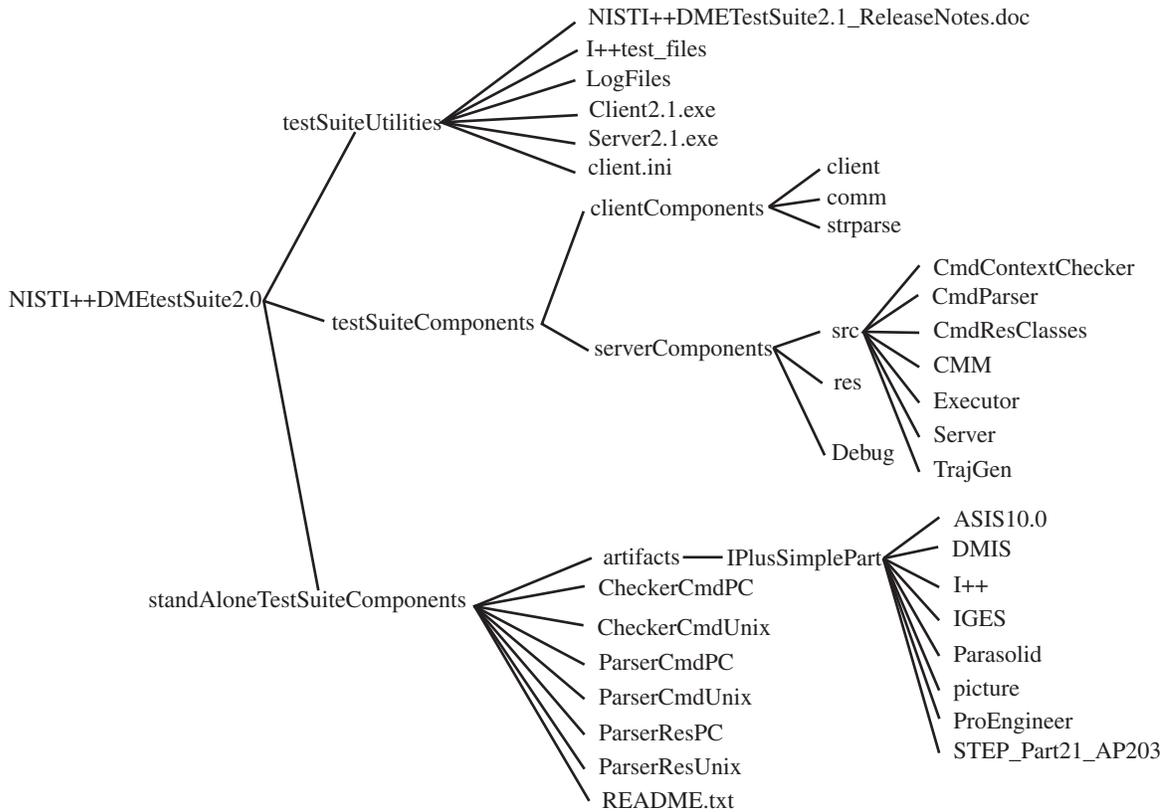


Figure 1: Directory structure for the NIST test suite version 2.1

## 2.2 Running the server-side and client-side utilities

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

NISTI++DMETestSuite2.1\testSuiteUtilities\Client2.1.exe  
NISTI++DMETestSuite2.1\testSuiteUtilities\Server2.1.exe

These executable files, client.exe and server.exe, are for testing server-side and client-side implementations of the I++ specification, respectively. For information on how to run client.exe, see Section 7.1. For information on how to run server.exe, see Section 8.1.

## 2.3 Using the test cases

The test cases currently consist of two items, 1) test files of I++ command strings and 2) a test artifact. The test files and files describing the artifact are included in the test suite and are located in the following directories:

```
NISTI++DMETestSuite2.1\testSuiteComponents\clientComponents\I++test_files
NISTI++DMETestSuite2.1\testSuiteComponents\clientComponents\artifacts\IPlusSimplePart
```

How to use the test files is described in Section 7.4 and how to machine and use the "simple" artifact is described in Section 7.5.

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 major 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.

## 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 6.2), 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 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

This test suite is currently designed to *facilitate* the creation of I++ DME Specification-compliant implementations and not to provide formal compliance tests. Informal interoperability testing is also currently in progress. The test suite currently facilitates both client and server side implementations and we suggest that it be used in the following three ways:

1. provide utilities that can be run against an implementation to validate correct performance and help speed up debug efforts
2. make 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. provide a benchmark interpretation of the specification that will be used to clarify and disambiguate the specification itself

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

### 4.1 Using the executables to facilitate implementation development

This type of use is described in Section 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, command and response string parsers that can be used by implementors 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 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 implementors 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.

### 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. Undoubtedly, as we develop implementations and use the test suite, weaknesses within the specification will be revealed. Our hope is that the specification writers will correct these weaknesses as they become known during the development and testing phase before the specification is released as an international standard. This is currently happening but needs to continue in earnest.

## 5 Using the test suite for conformance and interoperability testing

It is NIST's goal to have this test suite be expanded to constitute conformance and interoperability tests, *i.e.*, the components of this expanded test suite will be used to test the conformance of any single implementation of the specification and finally to support interoperability tests on pairs of implementations. Which is to say that the test suite provided is not currently accompanied by any test metric, test results analysis tools, or test procedures. More importantly, we have not defined enough test cases, providing sufficient coverage, to qualify the test suite as suitable for conformance.

On the other hand, we do not intend to create *formal* tests, either for conformance or interoperability. We believe that this is really more than is needed by the users of CMM systems. It is essential to point out that *perfect* conformance and interoperability can *never* be achieved, due to the inability of test suite to identify every possible misinterpretation of any interface language specification, and also due to the fact that it is practically impossible (without a formal definition language) to define a completely unambiguous specification.

Though we are not currently providing conformance tests, we do include a preliminary set of test cases (see Section 7.4) for facilitating both server-side and client-side implementation development. The client-side test cases consist of a test artifact 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. One of the server-side components is a single test case for responses. This test case always produces correct responses, but we have placeholders for adding more test cases which can create incorrect responses of various types. Our client-side utility also produces a log file to facilitate debugging and ultimately to use for testing compliance.

In an earlier version of the client-side utility for a prior DME specification development effort, the client-side utility was used to run a DME command file on a coordinate measuring machine (CMM) located at a facility in England (LK Metrology, Ltd.) remotely from NIST in Gaithersburg, Maryland, USA. For that demonstration, a user-controlled pan/tilt/zoom camera was integrated into the environment with a web-based video server to allow the inspection program to be viewed from NIST. We envision this capability to be particularly useful for future interoperability testing, as well as for conformance testing.

## 6 Introducing the components of the test suite

### 6.1 The two sets of components within the test suite: client-side components and server-side components

The test suite currently consists of two separate sets of components.

- Client-side components for facilitating the development of server-side (*i.e.*, the equipment side of the interface) implementations. Table 2 lists all the client-side components and gives a brief description of the operation of each component.
- Server-side components for facilitating client-side (*i.e.*, the application software side of the interface) implementations. Table 3 lists all the components of the server-side components and gives a brief description of the operation of each component.

The operation of these two sets of components, both as a test suite and as actual implementations, is illustrated in Figure 2. 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 implementors 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.

Table 2: A list of all the client-side components along with a brief description of what each does and where it is located

Client-side component	Description	Directory location
GUI front end	Provides a front end to the implementation developer that is easy to comprehend and use and also contains the "glue" code that integrates all client side components	NISTI++DMEtestSuite2.1\ testSuiteComponents\ clientComponents\client
TCP/IP socket read and write	Writes command strings and reads response strings from the specified TCP/IP channel	This component is currently integrated into the GUI front end code (client). Modularization is planned for a later version. However, it can currently be easily extracted from client.
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	A response parser is not required for operation of the current client-side utility, since no context checking is done, but a stand-alone response parser can be found in NISTI++DMEtestSuite2.1\ standAloneTestSuiteComponents\ ParserResPC
Client-side log file generator	The log file records time-stamped versions of all commands sent and responses received	Currently integrated into the GUI front end. Modularization is planned for a later version. Furthermore, 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	NISTI++DMEtestSuite2.1\ testSuiteComponents\ clientComponents\I++test_files
Test artifact	A 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 Section7.5)	NISTI++DMEtestSuite2.1\ standAloneTestSuiteComponents\ artifacts\IPlusSimplePart

## 6.2 Stand-alone components versus stripped-down components

All the components listed in Tables 2 and 3 are located in NISTI++DMEtestSuite2.1\testSuiteComponents\. This directory is intended to be used by those developers that 1) 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 components in their implementations. Some of the subdirectories do not contain source code, for example, the artifacts subdirectory. The source code files in NISTI++DMEtestSuite2.1\testSuiteComponents\ are compiled and linked into either the client-side or server-side utility. Therefore, they are stripped-down versions of the components, in the sense that they cannot be compiled and executed alone.

Several of the components have stand-alone versions with main() functions. These components are located in NISTI++DMEtestSuite2.1\standAloneTestSuiteComponents\. These will be helpful 1) to instruct an implementor how the component needs to be called within the implementor's code and 2) in certain cases, the stand-alone code can be used to test the validity of command and response strings files.

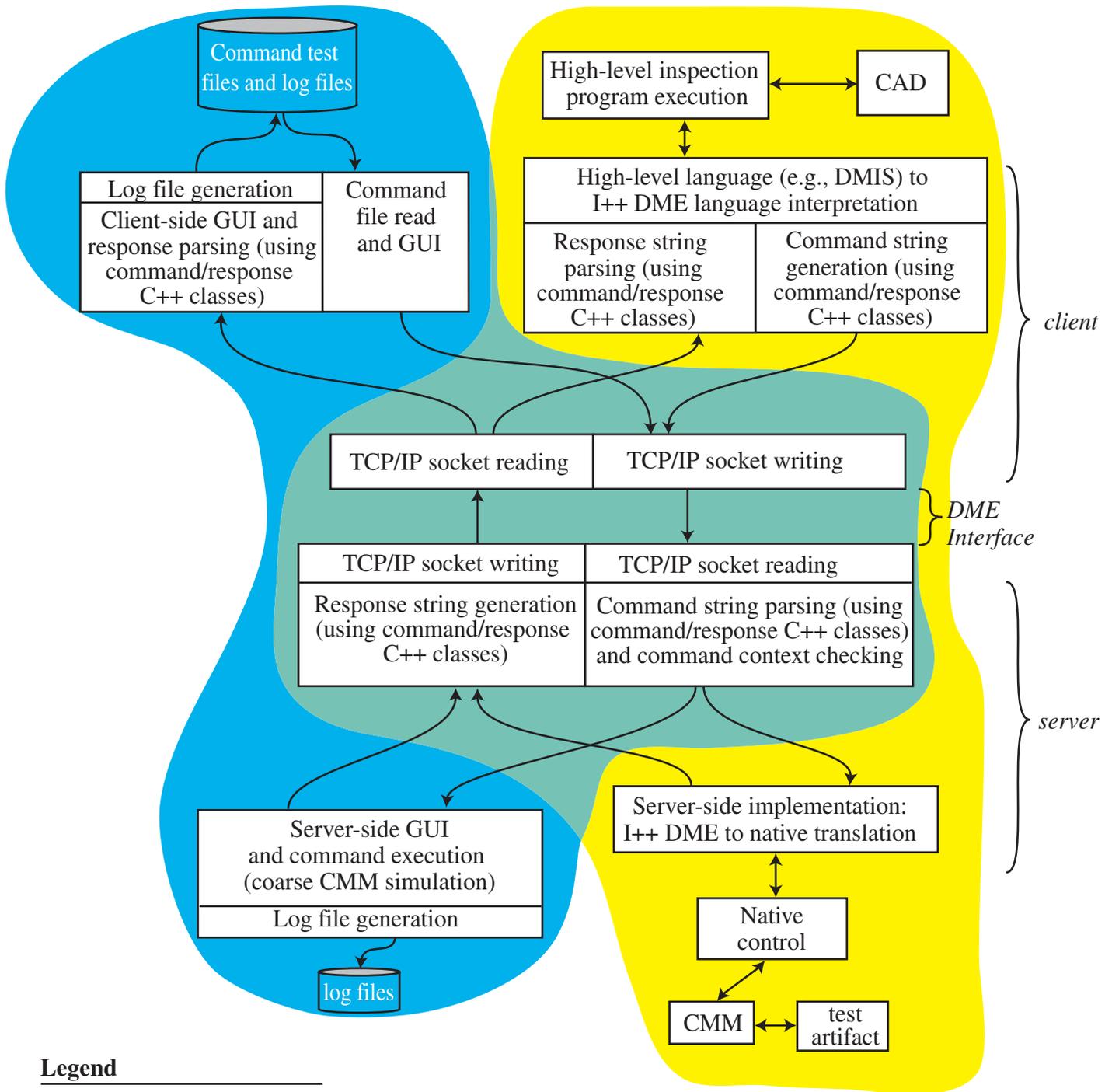


Figure 2: I++ DME Interface Specification Testing and Implementation Systems

Table 3: A list of all the server-side components along with a brief description of what each does and where it is located.

Server-side component	Description	Directory location
TCP/IP socket read and write	Reads command strings and writes response strings from the specified TCP/IP channel	Currently integrated into the GUI front end. Modularization is planned for a later version. Furthermore, it can be easily extracted from client.
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	NISTI++DMEtestSuite2.1\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	NISTI++DMEtestSuite2.1\testSuiteComponents\serverComponents\src\CmdContextChecker
Command executor (CMM simulator)	Executes commands and generates responses according to the test cases selected in the server-side GUI	NISTI++DMEtestSuite2.1\testSuiteComponents\serverComponents\src\Executor
Trajectory Generator	Provides a rudimentary trajectory generator for course CMM simulation	NISTI++DMEtestSuite2.1\testSuiteComponents\serverComponents\src\TrajGen
Server-side GUI front end	Provides a front end to the implementation developer that is easy to comprehend and use	NISTI++DMEtestSuite2.1\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 generate lists (queues) of commands and responses	NISTI++DMEtestSuite2.1\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	NISTI++DMEtestSuite2.1\testSuiteComponents\serverComponents\src\Server
World model	Contains data and methods for storing and maintaining information about the state of the CMM and its environment	NISTI++DMEtestSuite2.1\testSuiteComponents\serverComponents\src\CMM

### 6.3 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.

## 7 Using the client-side components for facilitating server-side implementations

Here's a summary of the operation of the client-side utility. The I++ DME commands are sent over a TCP/IP socket to an I++ DME compliant server. The server receives the commands, executes them, and returns the appropriate response back to the client-side utility via the socket. The client-side creates a time-stamped logfile of I++ DME commands sent and responses received over the socket.

The client-side components provide developers with a GUI allowing the developer easy control over a number of parameters, including the choice of the TCP/IP socket channel number, the choice of inspection programs (test files) to run, and the choice of whether to type in commands by hand, run test files *in toto*, or run test files line by line. The client-side GUI also generates a log

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	Keith Stouffer
Client-side utility	Keith Stouffer
TCP/IP socket read and write	Keith Stouffer
Command parser	Tom Kramer
Response parser	Tom Kramer
Client-side log file generator	Keith Stouffer
Test command files	Tom Kramer
Test artifact	John Horst
Command context checker	Tom Kramer
Command executor (CMM simulator)	Joe Falco
Trajectory Generator	Joe Falco
Server-side GUI front end	Keith Stouffer
Command and response C++ classes	Joe Falco, Tom Kramer, and John Horst
Server-side utility	Joe Falco, Tom Kramer

file that is helpful for debugging. The log file contains time-stamped versions of all commands sent and responses received for each test file run. The client-side utility checks that the response is consistent with the command sent for matching tag numbers between command and response. The client-side GUI contains a response parser, however, we have developed a new response parser that is more consistent with the command parser in the server-side utility (described in Section 8). This new response parser will be integrated into the client-side utility in a later version of the test suite.

In summary, the client-side components consist of the following components. Table 2 gives a description of each of the components listed here:

- GUI front end
- TCP/IP socket read and write
- Response parser
- Log file generator
- Test cases consisting of
  - Test files of I++ command strings (some with errors and some without)
  - A simple artifact consisting of a physical part and various CAD representations

## 7.1 Operating the client-side GUI front end

Figure 3 illustrates the client-side GUI. To start up the client-side GUI, navigate to the following directory

```
NISTI++DMETestSuite2.1\testSuiteUtilities
```

and double click on the file, Client2.1.exe. Select which type of test program to run (I++ DME or DMIS (Dimensional Measurement Interface Standard)). Click the "Set Driver File" button and navigate to the test program file you wish to run. When a DMIS file is selected, the file is run through an interpreter that converts the DMIS command to the appropriate I++ DME command(s). The DMIS mode is unsupported at this time. 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.

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

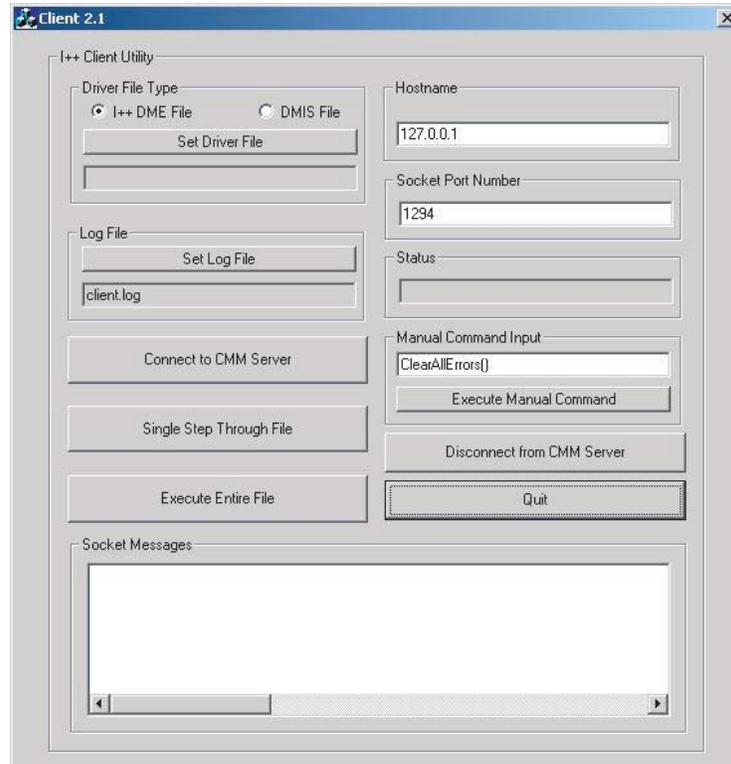


Figure 3: The client-side GUI front end

Client2.1.exe. If no client.ini file existed prior to starting up the GUI 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 GUI 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 GUI 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 1.30. 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. 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.

## 7.2 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 implementor 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 2.

## 7.3 Using the log file generation code

The log file generation code is modularizable, but not currently defined as a separate module. We plan to add this in our next version of the test suite.

However, if any implementors wish to extract the log file generation code themselves, they are able to. Simply access the file, `ServerDlg.cpp`, located in

```
\NISTI++DMETestSuite2.1\testSuiteComponents\serverComponents\src\Server
```

## 7.4 Using the command string test files

**Types of test files:** The command string test files consist of lines of 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 5.

Table 5: 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; AbortTest_WhileMoving.prg; AlignPart.prg; AlignTool.prg; ChangeTool.prg; ClearAllErrors.prg; Csy_test.prg; DisableUser.prg; EnableUser.prg; EndSession.prg; EnumAllProp.prg; EnumAllPropTest.prg; EnumProp.prg; EnumPropTest.prg; EnumTools.prg; FindTool.prg; Get.prg; GetCoordSystem.prg; GetCsyTransformation.prg; GetErrorInfo.prg; GetErrStatusE.prg; GetMachineClass.prg; GetProp.prg; GetPropE.prg; GetXtdErrStatus.prg; GoTo.prg; GoToTest.prg; Home.prg; HomeTest.prg; IsHomed.prg; IsUserEnabled.prg; OnMoveReportE.prg; OnPtMeasReport.prg; OnScanReport.prg; Probe1Test.prg; PtMeas.prg; PtMeasTest.prg; ReQualify.prg; ScanInCylEndIsPlane.prg; ScanInCylEndIsPlaneTest.prg; ScanInCylEndIsSphere.prg; ScanInCylEndIsSphereTest.prg; ScanInPlaneEndIsCyl.prg; ScanInPlaneEndIsCylTest.prg; ScanInPlaneEndIsPlane.prg; ScanInPlaneEndIsPlaneTest.prg; ScanInPlaneEndIsSphere.prg; ScanInPlaneEndIsSphereTest.prg; ScanOnCircle.prg; ScanOnCircleHint.prg; ScanOnCircleTest.prg; ScanOnLine.prg; ScanOnLineHint.prg; ScanOnLineTest.prg; ScanUnknownHint.prg; SetCoordSystem.prg; SetCsyTransformation.prg; SetProp.prg; SetTool.prg; StartSession.prg; StopAllDaemons.prg; StopDaemon.prg
Test files that contain some syntactical errors in command strings	numbers_error.prg, parserCmd_errors.prg
Test files that contain some logical or semantic errors in the command strings	checkerCmd_errors.prg
Test files that contain multiple correct command strings	numbers_ok.prg, all_ok.prg, all_res_ok.prg
Test files that contain commands for use on the test artifact	Simple.prg

We now describe the format and content of these files. This allows you to develop and be able to share your own test files with all other developers.

### 7.4.1 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 7.4.2 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 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 255 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 the digit zero <48> before an unsigned\_four\_digit\_integer. This makes CommandTags look like five-digit integers, but that appearance is irrelevant. 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  
10710 // error; first character not zero  
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>.

An argument is a number, a string, or a method\_identifier.

A simple\_method is a method name followed by a left parenthesis followed by either: a. a right parenthesis, or b. a number or a string followed by a right parenthesis.

A method\_identifier is one of the following:

- (a) a simple\_method
- (b) a method name followed by a period followed by a simple\_method
- (c) a method name followed by a period followed by a method name followed by a period followed by a simple\_method.  
Example: FoundTool.PtMeasPar.Speed()

A method name is the name of one of the methods defined in Sections 6.3, 11, and 12 of the I++ DME Interface Specification version 1.30. Method names are not enclosed in quotes.

Only those combinations of method names and arguments defined in Sections 6.3, 11 and 12 of the I++ DME Interface Specification version 1.30 may be used together.

## 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>

## 7.4.2 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 7.4.1, 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. It is suggested that the .txt file include every line of 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.

## 7.5 Using 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 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 not. The role of this simple artifact is therefore to expose obvious defects of functionality and interoperability early on in the testing process. At a later time we intend to introduce artifacts with a more complicated set of features.

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 implementor 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++DMETestSuite2.1\standAloneTestSuiteComponents\artifacts\IPlusSimplePart\`

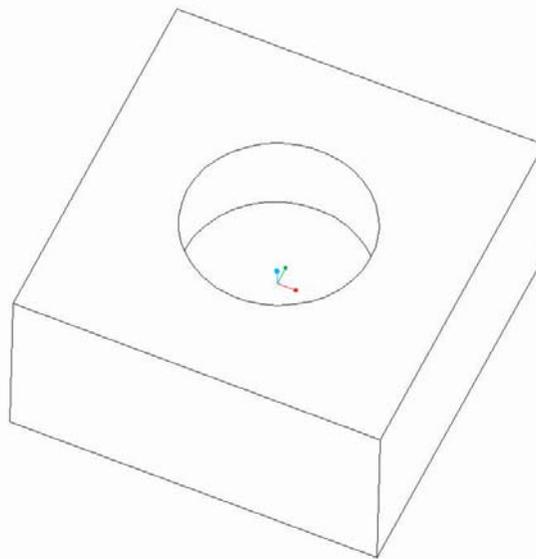


Figure 4: The simple test artifact

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++DMETestSuite2.1\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++DMETestSuite2.1\testSuiteUtilities\I++test_files`

## 7.6 Using the response parser code

The client-side utility executable contains a response parser. Another response parser that is more consistent with the command parser has been developed and tested, but not yet integrated into the client-side executable. Developers are encouraged to utilize this latter resource as a stand-alone parser for I++ response files to check that responses are parsable. We plan in our next version to integrate this parser into the client-side utility executable.

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

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserResPC\  
NISTI++DMETestSuite2.1\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).

### 7.6.1 Using the response parser as stand-alone support for server-side implementations

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

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserResPC\bin\parserRes.exe
```

Sample response files are in

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserResPC\test_files
```

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

```
NISTI++DMETestSuite2.1\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\all\_res\_ok.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 7.6.3 and 7.6.4. The test\_files subdirectory contains the test files:

- all\_ok.res
- all\_res\_ok.res
- error\_responses.res
- parserRes\_errors.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++DMETestSuite2.1/standAloneTestSuiteComponents/ParserResUnix
```

## 7.6.2 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.

Now we look at the syntax of individual response strings.

## 7.6.3 Format of 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 7.4.1. 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 7.4.1.

### 4. Response string length

The number of characters in a response string, including the `<13><10>` at the end, must not exceed 256. This differs from the length requirements for commands.

### 5. Numbers

The format of numbers is as described in section 7.4.1.

### 6. Strings

The format of strings is as described in section 7.4.1.

### 7. Tags

The format of tags is as described in section 7.4.1. 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 e (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. AlignTool Data

This is used only in response to an AlignTool 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", "MaxSpeed", "MinSpeed", "Accel", "MaxAccel", "MinAccel", "MinApproach", "Approach", "MaxApproach", "MinRetract", "Retract", "MaxRetract", "MinSearch", "Search", or "MaxSearch".
- (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", "MaxSpeed", "MinSpeed", "Accel", "MaxAccel", "MinAccel", "MinApproach", "Approach", "MaxApproach", "MinRetract", "Retract", "MaxRetract", "MinSearch", "Search", or "MaxSearch", 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

#### 7.6.4 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 7.6.3, 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`.

## 8 Using server-side components for facilitating client-side implementations

The server-side components of the test suite are listed and briefly described in Table 3. We will now describe how to use these components in your implementations of the I++ DME specification.

### 8.1 Operating the server-side GUI front end

Since the server-side GUI is set up to receive commands from the client-side, the user needs to set up and execute the server-side GUI first. To execute the server-side GUI, just double click the `Server2.1.exe` icon in the following directory.

```
NISTI++DMEtestSuite2.1\testSuiteUtilities
```

Once executing, this GUI will wait indefinitely for a connection from the client. Server-side GUI setup involves the following steps (refer to Figure 5 for a picture of the server-side GUI):

- 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 8.2.

- 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.

## 8.2 Server-side utility

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, and command execution (*i.e.*, coarse CMM simulation). The operation of these components is illustrated in Figure 2. It also 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 parsed and checked for a correctly formed tag and also to ensure that the tag number is not currently in use.
3. If the tag is legal, the command string is put in either the fast or slow queue.
4. 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.

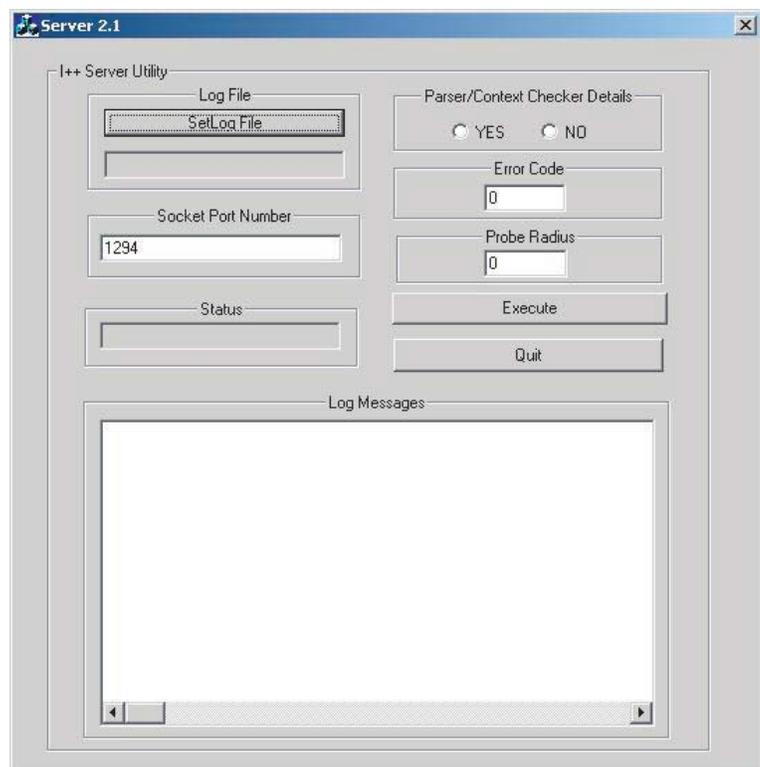


Figure 5: The server-side GUI front end

5. 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.
6. 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<sup>1</sup> for a command string.
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, AlignTool(), Home(), GoTo(), PtMeas(), or PtMeasIJK()<sup>2</sup>) 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 no 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 8.1), the log file will include information on commands received 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), both the I++ error type and a more detailed description of the error (if "yes" in the Parser/Context Checker Details option is selected) are logged

---

<sup>1</sup>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.

<sup>2</sup>actually PtMeasIJK() is not available in the server-side utility, since it appears to be identical to PtMeas() in the I++ DME spec version 1.30

- 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 8.8 on the executor.

### 8.3 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 7.2.

### 8.4 Using the command string parser

The command string parser is a separately defined software component with precisely specified interfaces. A stripped-down version of the parser has been integrated with the overall server-side utility (Section 8.2) and it also exists in a stand-alone version (with a main()). Developers are encouraged to utilize the stripped-down and stand-alone versions of the command parser in the following ways,

- In the stripped-down version as part of the server-side utility for facilitating client-side implementations
- In the stripped-down version for integration into an implementor'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 7.4.1 and command string files is in Section 7.4 under the heading, Command test file format.

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

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserCmdPC
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserCmdUnix
```

#### 8.4.1 Using the command parser as stand-alone support for client-side implementations

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

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserCmdPC\bin\parserCmd.exe
```

Sample command files are in

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\ParserCmdPC\test_files
```

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

```
NISTI++DMETestSuite2.1\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\all\_ok.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 7.4.1 and 7.4.2. The test\_files subdirectory contains the test files:

- all\_ok.prg
- all\_res\_ok.prg
- numbers\_ok.prg

- parserCmd\_errors.prg

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

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++DMETestSuite2.1/standAloneTestSuiteComponents/ParserCmdUnix
```

A Sun Solaris<sup>3</sup> 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".

## 8.5 CMM and tools related components

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 (world) into a separate set of source code files, world.h, world.cpp, and tools.h. These files are contained in the directory

```
NISTI++DMETestSuite2.1\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 implementors 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.

## 8.6 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 stripped-down version of the context checker has been integrated with the overall server-side utility (Section 8.2) 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 ???. Developers are encouraged to utilize the stripped-down and stand-alone versions of the command context checker in the following ways,

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

<sup>3</sup>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.

A description of the format of command strings is in Section 7.4.1 and a description of command string files is in Section ?? under the heading, Command test file format.

All files relating to the stripped-down and stand-alone versions of the context checker for the PC, respectively, described in this section can be found in

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\CheckerCmdPC
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\CheckerCmdUnix
```

A command context checker exists in both PC and UNIX (SUN systems) formats. 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 stripped-down and stand-alone components.

The source code defines functions in the checker class and defines a main function outside the checker class. It also includes documentation giving the rules for parsing the command strings for each I++ command. 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 1.30 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 reference pages in the source code reference both text and examples. Text references are given in parentheses. Example references are given in brackets. Other references are not enclosed. For example: Reference pages: 21 23 (35) [36] 63 means there is relevant text on page 35 and an example on page 36. Reference page 14 is not referenced because the print is too small and page 22 is not referenced since everything on it appears identically elsewhere.

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.

### 8.6.1 Using the command context checker as stand-alone support for client-side implementations

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

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\CheckerCmdPC\bin\checkerCmd.exe
```

Sample command files are in

```
NISTI++DMETestSuite2.1\standAloneTestSuiteComponents\CheckerCmdPC\test_files
```

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

```
NISTI++DMETestSuite2.1\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 file name such as, test\_files\all\_ok.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 7.4.1 and 7.4.2. The test\_files subdirectory contains the test files:

- all\_ok.prg
- checkerCmd\_errors.prg

For all\_ok.prg there is also a .txt file that explains the contents of the .prg file. For checkerCmd\_errors.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++DMETestSuite2.1/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"`.

## 8.7 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 `OnScanReport`, `OnPtMeasReport`, and `OnMoveReport` commands.

## 8.8 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 2.

## 8.9 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 implementors 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 2. 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.

## 9 Finding references to specific I++ DME commands in the I++ DME specification version 1.30

Table 6 gives an index to I++ DME commands found in release 1.30. The numbers refer to page numbers. The pages listed refer to both text and examples.

The command references are prioritized and categorized in Table 6 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 16 is never referenced because the print is too small and several of the command names in the figure are bogus. Page 24 is not referenced since everything on it appears identically elsewhere.

Table 6: Page number references to I++ DME commands given in the I++ DME specification version 1.30

<b>I++ DME 1.30 commands</b>	<b>Page numbers giving locations of prioritized and categorized references to each</b>
AbortE	23 26 (39) [40] 85
AlignPart	23, 28, 57, (84), 91
AlignTool	23 27 29 (50) 60 88 94
ChangeTool	23 27 29 (50) (68) 88 92
ClearAllErrors	23 26 (37) (39) (40) [41] (65) 66 85
DisableUser	23 27 (44) (47) (49) 88
EnableUser	23 27 [37] (44) 88
EndSession	17 19 23 26 (38) (67) 85
EnumAllProp	23 26 (42) (43) 85
EnumProp)	23 26 29 30 (42) [63] 85 94 95 96 97 98
EnumTools	23 29 (51) 92
FindTool	23 27 29 (49) (50) [63] 88 92 100
Get	23 27 [36] [40] [41] 45 (46) (47) (55) (56) (57) (60) 88
GetCoordSystem	23 28 (54) 90
GetCsyTransformation	23 28 (54) (70) 90
GetErrorInfo	18 23 26 (40) 85
GetErrStatusE	23 27 (46) 88 Called GetErrorStatusE on pages 27 and 88
GetMachineClass	17 23 27 (45) 88
GetProp	23 26 29 (42) 44 [63] 67 69 85 94
GetPropE	23 26 29 (42) 85 94
GetXtdErrStatus	23 27 (46) 88 Called GetXtdErrorStatus on pages 27 and 88
GoTo	23 27 [36] [40] (47) 48 (49) (56) (57) [61] [62] (68) 88
Home	23 27 [35] (44) 88
IsHomed	23 27 (44) 46 88 97
IsUserEnabled	23 27 (45) 46 88 100
OnMoveReportE	18 23 27 [37] (45) 68 88
OnPtMeasReport	23 27 (45) (47) 48 (56) [61] [62] (68) 88
OnScanReport	23 27 (71) 73 74 76 77 78 79 80 [82] 89
PtMeas	9 23 27 (45) (47) 48 (56) (57) [61] [62] 68 73 74 76 77 78 80 81 88
ReQualify	29 (58) 92 93 Called Qualify on pages 29 92 93
ScanInCylEndIsPlane	23 27 (80) 89
ScanInCylEndIsSphere	23 27 (79) 89
ScanInPlaneEndIsCyl	23 27 (77) (78) 89
ScanInPlaneEndIsPlane	23 27 (76) (77) 89
ScanInPlaneEndIsSphere	23 27 (75) (76) [82] 89
ScanOnCircle	23 27 (72) (73) [82] 89
ScanOnCircleHint	23 27 (72) [82] 89
ScanOnLine	23 27 (73) (74) 89
ScanOnLineHint	23 27 (73) 89
ScanUnknownHint	23 27 (75) [82] 89
SetCoordSystem	23 28 53 (54) [61] [62] 90
SetCsyTransformation	23 28 54 (55) [61] [62] (70) 90
SetProp	23 26 29 (42) (59) [63] 67 (68) 85 94
SetTool	23 27 29 (50) 88 92
StartSession	17 19 23 26 (38) 45 [61] (67) 85
StopAllDaemons	23 26 (39) 85
StopDaemon	(21) 26 [37] (38) 85