# A Comparison of the CMM-driver Specification Release #1.9 with the I++ DME-Interface Release 0.9

T. Kramer and J. Horst
Intelligent Systems Division
National Institute of Standards and Technology (NIST)
March 13, 2002

NISTIR 6863

## Disclaimer

## Acknowledgements

Partial funding for the writing of this paper was provided to Catholic University by the National Institute of Standards and Technology under cooperative agreement Number 70NANB7H0016.

## Abstract

This is a comparison of (1) the entire contents of *CMM-driver Specification Release #1.9* and (2) the parts of *I++ DME-Interface Release 0.9* that deal with the interface to a driver for dimensional measuring equipment.

## Keywords

CMM, dimensional, DMIS, I++, inspection, metrology, NIST, standard

# 1 Introduction

This comparison focuses on the interface between high-level inspection instruction execution and low-level inspection instruction execution. This interface is used between an executor of a high-level language (such as DMIS[1]) and a CMM[2] motion controller, for example. It would also be used between a CMM user interface and a CMM motion controller. This interface is called the "DME driver interface" in this report. The receiver of DME (dimensional measuring equipment) driver interface commands will be called the DME driver. The sender of DME driver interface commands will be called the application.

The items being compared are: (1) the entire contents of *CMM-driver Specification Release #1.9* and (2) the parts of *I++ DME-Interface Release 0.9* that deal with the DME driver interface. In the remainder of this report the version numbers are usually omitted for brevity, but all references to these documents mean these specific versions.

Much of *I++ DME-Interface* deals with matters that are not part of the interface. These are summarized below but are not analyzed.

The perspective of this report is that one or both documents being compared may become standards of a standards body (such as the International Organization for Standardization (ISO)) or may become *de facto* standards. In this event, the documents will have to stand on their own. To insure interoperability, the documents will have to be complete and unambiguous. Thus, this report does not take into account anything the authors or sponsors of the documents have said about their meaning and intent unless it is also written in the documents.

## 1.1 Contents of this Report

The remainder of this report includes:

1. general descriptions of the two documents.
2. a presentation of the important issues.
3. a broad comparison of the two documents.
4. a detailed comparison of the two documents.

## 1.2 Other NIST Documents

Other NIST documents relevant to this comparison (available on request to T. Kramer) include:

1. *Analysis of Dimensional Metrology Standards*; NISTIR 6847.
2. a proposed revision (from December 2000) of an earlier version of the *CMM-driver Specification*.
3. a comparison (from December 2000) of the low-level command set used in the NIST. DMIS interpreter with the commands from the *CMM-driver Specification*
4. an eight-page set of detailed comments (from November 2000) on the *CMM-driver Specification*.

1. DMIS = Dimensional Measuring Interface Standard
2. CMM = Coordinate Measuring Machine

# 2 General Descriptions

## 2.1 General Description of CMM-driver Specification

The *CMM-driver Specification Release #1.9* is a 44-page document that was developed by representatives (Lutz Karras [Carl Zeiss], Michel Penlae [Xygent], David Smith [LK], and Bill Wilcox [Brown & Sharpe]) of four companies working the area of metrology hardware and software.

The entire text of the *CMM-driver Specification* deals with a DME driver interface.

The *CMM-driver Specification* specifies a method of communicating and a set of commands.

Implementations of the *CMM-driver Specification* have been built and tested.

## 2.2 General Description of I++ DME-Interface

The *I++ DME-Interface Release 0.9* is a 62-page document that was developed by representatives (Hans-Martin Biedenbach [Audi], Josef Brunner [BMW], Kai Gläsner [DaimlerChrysler], Günter Moritz [Messtechnik Wetzlar], Jörg Pfeifle [DaimlerChrysler], and Josef Resch [Zeiss IMT]) of three automobile companies and two companies working the area of metrology hardware and software.

The *I++ DME-Interface* document specifies an object-oriented system. The focus of the document is on an object called the DME-Interface. This is an object, not an interface; to avoid confusion, we will call this object the DME-I. Page 9 of the document shows a system layout with the DME-I at the center. The layout includes a line joining the DME-I to an application. This line is the DME driver interface in which we are interested. The DME-I is responsible for all subsystems of the DME, so all commands and responses flow through the DME-I. Thus, almost all of the contents of the document deal with the interface in which we are interested.

The DME-I is modeled as an instance of an object class in a class hierarchy including the classes Server, DME, Cartesian CMM, and Cartesian CMM with Rotary Table. The DME-I has associated objects, the most important of which are tool changers, tools, and axes.

For the most part, the details of the how the DME-I is modeled can be ignored without creating any misunderstanding about the commands and responses the DME-I can handle. This report will say nothing further about the object modeling of the DME-I unless it is essential.

The *I++ DME-Interface* document appears to be incomplete. A number of DME methods which are on the diagrams (Picture 10 and Picture 11) are not included in the section (6.3.2) on DME methods, or in any other methods section, and, thus, are presumably not among the commands that may be sent by the application. These methods are: ChangeTool, FindTool, Tool, GoToPar, ABCGoToPar, PtMeasPar, and ABCPtMeasPar. Judging from the names ("Par" means parameter), the missing methods include, among other things, all the methods for setting and getting parameters. The examples of exchanges between application and DME driver (section 7.7

in particular) include calls to some of the missing methods. It may be that the authors intentionally deleted the methods from section 6.3.2 and did not get around to updating the rest of the document, but it seems much more likely that they simply did not complete section 6.3.2. In any event, in the absence of a description of what a command does, it is hard to compare its meaning to anything.

At NIST, we have not heard of any implementations of the *I++ DME-Interface* having been built.

## 2.3 Events and Daemons

The *I++ DME-Interface* includes (in section 4) the notion of an "event", used in a different sense from the usual meaning in computer programs that deal with events. The usual meaning is "anything of interest that happened". In the *I++ DME-Interface*, an event is a command or response of high priority. In connection with events, the *I++ DME-Interface* uses the notion of a "daemon". A daemon is a thread or subprocess in the DME driver that triggers events.

The *CMM-driver Specification* does not use events or daemons, but it does provide that some commands and responses have higher priority than others.

The *I++ DME-Interface* contains many commands whose names end with an upper case E. It seems intended (although nothing explicit regarding command names could be found in the text) that these commands are event commands and the DME driver is expected to generate events in response.

In several cases, the *I++ DME-Interface* contains pairs of commands whose names differ only by one having an E at the end and the other not. The descriptions of these commands are identical. It seems intended that the command without the E is a normal command, while the command with the E is an event command.

The remainder of this report does not discuss events. Daemons are not described in detail in this report but are mentioned where the *I++ DME-Interface* uses them.

# 3 Issues

This section presents issues involved with the DME driver interface. All but the last two of these are used in section 4 for comparing the *CMM-driver Specification* with the *I++ DME-Interface*.

## 3.1 Scope

Every interface specification should include a statement of scope describing the domain of activity to which the specification is intended to apply. The scope should indicate the types of equipment with which the specification is expected to be usable.

It will be helpful if the scope includes mention of the execution models the standard is intended to support, since some execution models may require more information than others.

For a DME driver interface, the scope should include at a minimum:

1. the activity of dimensional inspection by probing points.
2. the control of CMMs.
3. touch trigger probes.

Other types of dimensional measuring activities and equipment might also be covered, such as:

1. scanning probing.
2. control of non-contact optical metrology systems, such as laser scanners, theodolites, and photogrammetry equipment.
3. non-contact capacitance probes.

## 3.2 Purpose

Every interface specification should include a statement of purpose describing what capabilities the specification is intended to support. If the interface specification does not state its goals, it is impossible to determine if it achieves its goals.

For a DME driver interface, the purpose should include supporting interoperability and supporting DMIS.

## 3.3 Communications

A DME driver interface specification might include a specification of communications methods, or communications methods might be the subject of a separate specification. If a communications method is included, it should be one that is readily available.

## 3.4 Messaging Protocol and DME Driver States

A messaging protocol (for any message-based interface specification) is a set of rules for conducting conversations between the sender and receiver of interface commands and responses. This protocol governs which party can send which commands or responses when and what the other party should do about it. Very different behavior may occur in systems using the same commands and responses (with well-defined semantics) but a different messaging protocol.

A messaging protocol is not provided by a specification of communications methods. Rather, the messaging protocol must be built on top of communications methods. A messaging protocol should be included in a DME driver interface specification.

Most provisions of a messaging protocol may be expected to be involved with startup, shutdown, and error handling. Normal steady state operations are likely to be covered by a small portion of the protocol.

For a messaging protocol to be specified, some method is required for characterizing the condition of the receiver. The most straightforward way of doing this is to have receiver state variables and possible values for those variables. Different sets of messages may be sent according to the values of the receiver's state variables. State variables of a DME driver might include, for example: readiness (with values of unready, ready, paused, or erred) and probe_inhibited (with [boolean] values of true or false).

## 3.5 Execution Model

The execution models that a DME driver interface intends to support need to be explicitly stated in a DME driver interface specification. Two separate aspects of execution models that should be covered are:

1. whether the DME driver (a) accepts only one command at a time or (b) accepts multiple commands, puts them in a queue, and manages the queue. If queue management is expected, commands for queue management are needed.
2. whether the DME driver and application (a) use one-time messaging or (b) use cyclic messaging.

The first of these is necessarily reflected materially in the messaging protocol. For example, typically, in case (a) the application should not send a new command until the DME driver returns a response saying the last command is done, while in case (b) the application may send a new command as soon as the last command is acknowledged as having been received.

The second of these is also reflected in the messaging protocol, but may be handled simply. In cyclic messaging, the application and the DME driver run on a fixed clock cycle and each sends a message every cycle. A message may often be identical to the preceding one, except possibly for an id number. A messaging protocol for one-time messaging may readily be used in a cyclic messaging system by considering only messages that are not identical to their predecessors.

There may be other aspects of execution models that are also important; this is not presented as an exhaustive list.

## 3.6 Responsibility for Accuracy

A very important issue for users is: who is responsible for the accuracy of measurements taken on a DME. Is it the DME vendor (who would implement the DME driver), or is it the vendor of the high-level language executor (the application). If the interface allows enough data to pass across it, compensations for point data can be made by the application. If the application makes the compensations, the application vendor is primarily responsible for the accuracy of point data. If the interface does not allow enough data to pass across it to support compensations, compensations can only be made on the DME driver side, and the DME vendor is responsible.

The two major types of compensation (not regarding calibration as compensation) are volumetric compensation (for measured machine imperfections in specific parts of the work volume) and thermal compensation (for changes in the machine shape from thermal expansion or contraction). Other types of compensation exist, and some systems may use them.

This issue is also important for third-party software vendors whose software performs corrections on data points. If the DME driver interface does not support the exchange of the data they need, they cannot market their products as third parties. They can still attempt to sell their products to vendors to work hidden in the DME driver, but they cannot sell their products directly to users.

Additionally, this issue is important for certain non-contact metrology systems which are known to put uncompensated point data on the interface. This is particularly important in systems where contact and non-contact CMMs work together.

It is important to realize that this issue only regards point data. It does not regard analyzed data, such as feature data (the center and diameter of a circle, for example). The creation of feature data is outside the scope of the DME driver interface. Feature data is normally created either by the application (which may well call on third party software) or by a separate analysis package. If the accuracy of feature data is an issue, both point data and the software used to fit features to points must be examined.

## 3.7 Command Set (Coverage and Support for DMIS)

The command set provide by a DME driver interface should be complete within its stated scope and purpose.

DMIS is the only standard high-level language for control of dimensional measuring equipment. A DME driver interface should support that standard. In particular, a DME driver interface should include functionality that makes it possible to carry out all DMIS commands.

Where equipment vendors believe DMIS functionality is unreasonable (there are such areas), and where I++ members find DMIS in need of improvement, proposals for modifying DMIS should be considered. Consideration of proposals for changes to DMIS should have full international participation.

## 3.8 Message Syntax

The syntax of messages should be completely and unambiguously defined down the individual character level. By syntax we mean what characters may be used, how they may be combined to form tokens (words, numbers, etc.) and how tokens and punctuation may be combined to form messages.

## 3.9 Semantics

The meaning (semantics) of the commands and responses must be fully described in a DME driver interface specification. If the meanings are not fully specified, the behavior of different implementations cannot be expected to be very similar.

Semantics cannot yet be captured fully in any formal language. Unambiguous natural language descriptions of meaning are required to accompany any formal specification.

## 3.10 Language

It is useful if an interface specification is written in a standard information modeling language rather than in some ad hoc format. The information model (including the accompanying text) contains all the semantics of the interface. Tools exist for compiling information models written in standard languages, providing a valuable check on model syntax and completeness. Three standard information modeling languages are: EXPRESS, CORBA IDL (Common Object Request Broker Architecture Interface Definition Language), and XML (eXtensible Markup Language) schema. C++ classes may also be considered to be an information modeling language. If a specification is written in an ad hoc format, it cannot be checked automatically for syntax and completeness errors.

If an information modeling language is used, a method of writing instances of the entities in the model is needed. The STEP (STandard for the Exchange of Product model data) standard and XML both provide a file format. With C++ classes, no standard method of writing instances is provided, and the user must invent one. Here again, a standard language has a large advantage over an ad hoc language, in that automatic tools exist that will read and write files of instances and will check syntax.

## 3.11 Interoperability

A DME driver interface should support interoperability of dimensional measuring equipment.

In particular, a DME driver interface should make it possible for a high-level controller to control similarly constructed dimensional measuring equipment using different low-level controllers without changing more than configuration files and operating parameters. Standardizing the DME driver interface and conforming to that standard will provide this capability.

Also in particular, a DME driver interface should make it possible, under the same conditions as above (same high-level controller, similar equipment, different low-level controllers), to get nearly identical behavior and inspection results when executing the same high-level program. Standardizing the DME driver interface and conforming to that standard will also provide this capability.

As described in section 3.9 and section 3.4, having clear semantics and a clearly defined messaging protocol in the DME driver interface specification is required for interoperability.

It is desirable that it be possible to execute the same high-level program on different high-level controllers and get nearly identical behavior and inspection results. Standardizing the DME driver interface and conforming to that standard will make achieving this capability much easier but is not sufficient. Conformance of the high-level controller to the high-level language is also required; that is out of the scope of this document.

## 3.12 Specification Development

### Single Specification

The world-wide market for dimensional measuring equipment is too small to benefit from competing DME driver interface specifications. Competing specifications lead, obviously, to reduced interoperability. This seems likely to be bad for both users and vendors of dimensional measuring equipment. Vendors and users worldwide should work towards agreement on a single standard DME driver interface specification.

To be an international standard, the DME driver interface should be made an ISO standard. If the DME driver interface is made a national standard of two or more countries before being considered by ISO, differences in the national versions will inevitably arise that will make it more difficult to agree on a single ISO version.

### Specification Validation

Like a computer program, a technical standard such as the DME driver interface cannot be expected to be correct until it is thoroughly tested and debugged. If anyone suggested writing a large new computer program by committee consensus and then selling it without compiling it and without testing it, he or she would be considered marginally insane. Nevertheless, standards are usually developed by committee consensus and often become national or international standards with little or no testing.

Any DME driver interface should be thoroughly tested and debugged before it is agreed to use it commercially and before it becomes a standard of any sort.

### Conformance

After a DME driver interface is standardized, the issue of conformance to the standard becomes important, but that is outside the scope of this report.

# 4 Comparison of Broad Areas

This section provides a broad comparison of the *CMM-driver Specification* and the *I++ DME-Interface*.

In the tables in this report, section numbers of the *CMM-driver Specification* and the *I++ DME-Interface* are included inside parentheses, (3.2.5) for example. An X in a table means the item does not exist. Comments in the tables are given in helvetica type.

## 4.1 Scope

The scope of the *CMM-driver Specification* is given on five lines. These lines provide that touch trigger sensors, spherical probe tips, 3 axes CMMs, and 2 axes probe heads are in scope.

The scope of the *I++ DME-Interface* is cartesian coordinate CMMs.

Neither document discusses what execution models are intended to be supported.

## 4.2 Purpose

Both the *CMM-driver Specification* and the *I++ DME-Interface* give no purpose at all. Neither makes any explicit commitment to supporting interoperability and neither makes any commitment to supporting the functionality available within DMIS or any other high-level language.

Both need to have an explicit statement of purpose added.

## 4.3 Communications

The *CMM-driver Specification* states briefly that sockets should be used.

The *I++ DME-Interface* states that TCP/IP sockets will be used for communications. Section 3 of the document (e.g., picture 4 in section 3.1), however, indicates that communications is a layer separate from the client (application) and server (DME driver), so that communications might be changed in the future.

The two documents agree on communications.

It might be preferable if specifying the method for communicating were separate from the rest of the DME driver interface specification. This would allow one to be changed without affecting the other. A specification of both is needed in order to build a working system, of course, so this is not a major consideration. Since communications is an area of agreement between the two documents, it may be best to leave it alone.

## 4.4 Responsibility for Accuracy

The two documents differ regarding the responsibility for accuracy.

The *CMM-driver Specification* provides commands that allow compensation to be made on either side of the interface. We have not tried to determine if these are adequate to serve the needs of third-party software vendors.

The *I++ DME-Interface* states specifically in section 1.4 that “the DME vendor is responsible for the accuracy of his measurement equipment, in the sense that all necessary functions related to the equipment accuracy have to be implemented in the [DME-I]”. This implies that there should be no interface functions intended to support making accuracy compensations outside the DME-I, and none are provided.

One way the difference between the two documents might be resolved is to provide in the specification that there should be two conformance classes, one that includes the commands necessary to do compensation in the application, and one that does not. A DME vendor would declare in which conformance class the vendor’s system lies. The user could decide which class to buy.

## 4.5 Command Set (Coverage and Support for DMIS)

The command set provided by the *CMM-driver Specification* appears to be adequate to support almost all of the functionality expected of a CMM with a touch trigger probe. It also appears to be adequate to support a large portion of DMIS.

As noted earlier, the command set provided by the *I++ DME-Interface* appears to be incomplete. Even if all the methods mentioned anywhere in the document were added to the protocol (section 6), important pieces would be missing, such as access to almost all configuration data.

If and when other types of equipment and sensors are added to the scope, additional commands will be needed to support them.

## 4.6 Messaging Protocol and DME Driver States

The messaging protocols of the *CMM-driver Specification* and the *I++ DME-Interface* are similar. Neither one, however, is adequately defined. Neither one has DME driver states or any other method of specifying which messages may be sent under what conditions.

In both messaging protocols, the usual exchange starts when the application sends a command, the application must then wait for an acknowledgement from the DME driver that the command has been received before sending another command. On receiving a command, the DME driver sends an acknowledgement and puts the command on its queue of commands to execute. When it finishes executing a command, the DME driver sends a done message. In the *CMM-driver Specification*, the done message often includes returned data. In the *I++ DME-Interface*, the done message contains no data (done is just plain done), and if data is to be returned, it is returned in a separate message sent before the done message. The name of the done message in the *I++ DME-Interface* is “ready”, but all it means is that execution of the indicated command is over (either because it was completed successfully or because it was aborted). In the *CMM-driver Specification*, done means completed successfully.

Both documents provide that some messages have higher priority than others. Dealing with commands of different priority in a queuing system is very tricky. Both documents are quite vague about how this is to be done and need major improvement. Defining DME driver states should be useful in connection with handling commands with different priorities.

The *I++ DME-Interface* specifies that, except for responses identified as events, the DME driver must handle commands in the order received, and must send all messages responding to one before sending any data, error, or done messages responding to the next one. Acknowledgement messages, however, may be sent at any time, and actual execution of commands may overlap.

The *CMM-driver Specification* specifies only that the application must wait until it receives an acknowledgement or error message from the DME driver before sending another command. Additional responses to a command may be sent in any order, and commands are not required to be carried out in order. The *CMM-driver Specification* may intend that certain commands (such as MOVE_AXIS) be carried out in order. If so, it should be changed to make such cases explicit. If not, it should include a warning to application builders.

The handling of command identifiers differs between the two documents. In the *CMM-driver Specification*, the DME driver assigns them. In the *I++ DME-Interface*, they are assigned by the application. Having them assigned by the application is preferable, stemming from the fact that the application is the party that creates the command.

The *I++ DME-Interface* goes a little beyond the *CMM-driver Specification* in the case of commands for which multiple responses are expected. For such commands, the *I++ DME-Interface* specifies that a daemon should be created that causes messages to be sent periodically. Commands for stopping daemons are provided.

The *I++ DME-Interface* provides a number of examples of message "conversations" between the application and the DME driver. This is a very useful feature of the *I++ DME-Interface*. The *CMM-driver Specification* provides no examples.

## 4.7 Message Syntax

The *CMM-driver Specification* syntax specification is loose and needs tightening. It allows characters that are not recognized in the syntax to be included in messages. This is very unwise in that it allows errors to slip through and allows implementations to hide proprietary messages inside DME driver interface messages. The occurrence of an unrecognized character should always be considered an error. The *CMM-driver Specification* does not define types of data and how data of each type should be written in messages; it should do that.

The *I++ DME-Interface* syntax specification does not explicitly allow characters not in the syntax, but is otherwise slightly looser. In addition to not defining data types, it provides no description at all of numbers or command arguments, which the *CMM-driver Specification* does provide.

A good syntax specification would be written in a syntax specification language such as Extended Backus Naur Format (EBNF). That would prevent arguments about what syntax is legal, since such languages specify syntax down to the individual character level.

### 4.8 Semantics

As noted in the issues, complete semantics are essential for interoperation. Both specifications need improvement in semantics.

The *CMM-driver Specification* is very weak on semantics. It rarely devotes more than a phrase to the meaning of a command. Descriptions of other items are often sketchy. Some important items are given no description at all.

The *I++ DME-Interface* is fair on semantics. Certain commands, such as PtMeas, are thoroughly described, while others, such as Abort, have brief, incomplete descriptions.

### 4.9 Language

The notation used in the *CMM-driver Specification* for describing syntax is non-standard. The document does not explicitly describe the notation. It would be useful if it did so.

The *I++ DME-Interface* uses an ad hoc method of describing the format of messages, but uses C++ function format for most of the contents of messages.

Neither document takes full advantage of modern information modeling techniques, but it may make reaching agreement on a single standard easier if ad hoc methods continue to be used.

### 4.10 Execution Model

Neither the *CMM-driver Specification* nor the *I++ DME-Interface* describes explicitly the execution models that it intends to support. Both clearly expect the DME driver to maintain a queue of pending commands, but neither says much about queue management. Neither mentions cyclic messaging.

**Table 1: Broad Areas**

| Issue | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| Scope | (3.1)<br>touch trigger sensors, spherical probe tips, 3 axes CMMs, and 2 axes probe heads are in scope | Not explicit, but implied by figures and C++ code, particularly Pictures 10 and 11.<br>cartesian coordinate CMMs are in scope |
| Purpose | none explicitly stated | none explicitly stated |
| Communications | (3.2.1)<br>use sockets | (3.1)<br>use TCP/IP sockets |
| Responsibility for Accuracy | (no specific section)<br>allows either application side or DME driver side (DME vendor) to be responsible for accuracy | (1.4)<br>only DME vendor may be responsible for accuracy |
| Command Set (Coverage and Support for DMIS) | (4) twenty-one pages.<br>complete or almost complete for CMMs with touch trigger probes, supports much of DMIS | (6) twenty pages.<br>incomplete |
| Messaging Protocol and DME Driver States | (3.2)<br>messaging protocol not clearly described.<br>no DME driver states provided. | (6.1 & 6.2)<br>messaging protocol not clearly described.<br>no DME driver states provided. |
| Message Syntax | syntax needs tightening | syntax needs tightening |
| Semantics | needs major improvement | needs improvement |
| Language | ad hoc | ad hoc with C++ |
| Execution Model | not explicit, queuing expected | not explicit, queuing expected |

KEY: In this table and all other tables in this report, section numbers of the *CMM-driver Specification* and the *I++ DME-Interface* are included inside parentheses, (3.2.5) for example. An X in a table means the item does not exist. Comments in the tables are given in helvetica type.

## 5 Detailed Comparisons

This section provides a detailed comparison of the *CMM-driver Specification* and the *I++ DME-Interface*. Each subsection focuses on one topic. Most of the topics are types of commands. Each subsection has text at the beginning and a table at the end.

In several cases, the *I++ DME-Interface* contains pairs of commands with identical names but one has an empty argument list and the other has a non-empty argument list. This (alas) is standard practice in object-oriented languages. These are different commands. The command with the empty argument list is generally a “get” command that gets the value of something, while the one with the non-empty argument list is a “set” command that assigns a value to the same thing.

### 5.1 Administrative Commands

Administrative commands deal with starting to use the DME driver interface, stopping using it, dealing with errors, and dealing with a queue of unfinished commands, some of which may be executing, and others of which have not yet been started.

Both documents assume that the DME driver may maintain a queue of accepted commands.

The *CMM-driver Specification* is weak regarding queue handling. The *I++ DME-Interface* is better but needs further tightening.

The *I++ DME-Interface* defines the notion of a session. The *CMM-driver Specification* does not. Session is a useful notion and should be defined.

**Table 2: Administrative Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| stops executing one currently executing command | ABORT (4.1.1) | StopDaemon (6.3.1.3) |
| stops executing all currently executing commands | | StopAllDaemons (6.3.1.4) |
| removes pending commands from the queue | | Abort (6.3.1.5) It is not clear whether commands that have been started are considered pending, or only commands that have not been started, or both. |
| clears any and all errors | CLEAR_ERROR (4.1.2) | ClearAllErrors (6.3.1.6) |
| stops accepting DME driver interface commands | CLOSE_DRIVER (4.1.3) | Disconnect (6.3.1.2) |
| pauses or resumes accepting DME driver interface commands | CMM_DRVR (4.1.4) Proprietary commands may be sent while accepting DME driver interface commands is paused. | X |
| obtains status of DME driver | GET_STATUS (4.1.6) | ErrStatus (6.3.2.11)<br>ErrStatusE (6.3.2.12) |
| obtains more detailed status of DME driver | X | XtdErrStatus (6.3.2.13) |
| begins accepting DME driver interface commands | INIT_DRIVER (4.1.5) | Connect (6.3.1.1) |
| resets all parameters to default values | RESET (4.1.7) | X |

## 5.2 Commands that Change Responses

The *CMM-driver Specification* has mostly fixed specifications for what information should be included in responses to commands and provides few commands that change what should be included.

The *I++ DME-Interface* has more modifiable specifications for what information should be included in responses to commands and provides more commands that change what should be included.

**Table 3: Commands that Change Responses**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| specifies what should be reported after executing GoTo | X | OnEndOfGoToReport (6.3.2.6) types of “properties and “methods” need to be better described |
| specifies what should be reported on executing PtMeas | SET_PARAMETER (TOUCH_REPORT) (4.9.10) | OnPtMeasReport (6.3.2.7) |
| specifies what should be reported while executing GoTo | SET_PARAMETER (TIMED_POSITION_REPORT) (4.9.9) | OnMoveReportE (6.3.2.8) |
| specifies what should be reported when the joystick is enabled and the user uses the joystick to move the machine | | OnManualMoveReportE (6.3.2.9) |

## 5.3 Free Space Motion Commands

The free space motion commands of the two documents are very similar, except that the *I++ DME-Interface* provides a command to ask if the machine is homed, which is not provided by the *CMM-driver Specification*.

**Table 4: Free Space Motion Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| gets the current position of axes | GET_POS (4.3.1) | Get (6.3.3.13)<br>GetE (6.3.3.14) |
| moves to home position | HOME (4.3.2) | Home (6.3.2.1) |
| moves to specified position | MOVE_AXIS (4.3.3) | GoTo (6.3.3.15) |
| determines if the machine is homed | X | IsHomed (6.3.2.2)<br>It is not clear whether a machine is considered to be homed only when it is at home position or at all times during a session following successful execution of the Home command. |

## 5.4 Probing Commands

As noted earlier, commands for activating tools have been omitted (apparently inadvertently) from the *I++ DME-Interface*. It is not clear why there is no explicit command to measure a point manually. The EnableUser command (6.3.2.3) is certainly intended to allow manual measurements, but the OnManualMoveReportE command (6.3.2.9) does not seem to provide a means to distinguish way points from hit points.

**Table 5: Probing Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| manually measures a point | PROBE_MAN (4.4.1) | X |
| automatically measures a point | PROBE_TO (4.4.2)<br>Weak semantics. | PtMeas (6.3.3.16)<br>Strong semantics. |
| activates named probe tip | ACTIVATE_SENSOR (4.5.1) | X |
| activates or deactivate current probe tip | INHIBIT_PROBE (4.5.2) | X |

## 5.5 Multiple Carriage Commands

The *I++ DME-Interface* does not deal with multiple carriages. Section 1.6 says they will be added between releases 1.0 and 1.1.

**Table 6: Multiple Carriage Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| turns on or off the automatic coupling of carriages | COUPLING (4.2.1) | X |
| gets the transformation that controls how two carriages will be coupled when they are coupled | GET_COUPLING_DATA (4.2.3) Semantics unclear. | X |
| sets the transformation that controls how two carriages will be coupled when they are coupled | SET_COUPLING_DATA (4.2.2) Semantics unclear. | X |

## 5.6 Calibration Commands

The *CMM-driver Specification* provides three calibration commands. These make it possible to implement DMIS commands for calibration.

The *I++ DME-Interface* provides no calibration commands. Calibration commands do not make it possible to perform compensations outside the DME driver, but their existence would be a little contrary to the notion that the DME vendor should be responsible for accuracy (see section 3.6 and section 4.4). Presumably, the DME driver is expected not to be ready to go until all calibrations are complete. It would be pointless to include a command to do something that must already have been done.

**Table 7: Calibration Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| calibrates a tip on the current sensor assembly | CALIBRATE (4.5.3) | X |
| defines the shape of the artifact used for calibrating sensors | DEF_CALIB_ARTIFACT (4.5.5) | X |
| starts a calibration sequence | START_CALIB_SEQUENCE (4.5.4) | X |

## 5.7 Miscellaneous Commands

This section includes those commands for which there is only one or two of a type.

**Table 8: Miscellaneous Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| loads and activates a sensor assembly | LOAD_ASSM (4.6.1) | X |
| unloads a sensor assembly | UNLOAD_ASSM (4.6.2) | X |
| disables joystick control | SET_AUTO_MODE (4.11.1) | DisableUser (6.3.2.4) |
| enables joystick control | SET_MAN_MODE (4.11.2) Control by command works while in manual mode. | EnableUser (6.3.2.3) |
| determines whether joystick control is enabled | X | IsUserEnabled (6.3.2.5) |
| gets data about a rotary table | GET_TABLE_DATA (4.12.1) | X |

## 5.8 Coordinate Systems

The *CMM-driver Specification* requires the use of machine coordinates at all times (corrected if correction is available). The point whose coordinates are given, however, may always be either (a) a point in a fixed location with respect to the ram or (b) the point at the center of the probe tip. The *CMM-driver Specification* does not allow any other coordinate systems for axis data in commands or responses. The *CMM-driver Specification* does allow the coordinate system for the digital read out to be changed and the coordinate system for the joystick to be changed.

The *I++ DME-Interface* allows the application to select one of three coordinate systems: the MachineCsy (machine coordinate system), the MultipleArmCsy (multiple arm coordinate system), or the PartCsy (part coordinate system). Set and get methods are provided for the part coordinate system.

**Table 9: Coordinate Systems**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| sets the coordinate system with respect to which digital read-out (DRO) coordinates are given | SET_DRO_MATRIX (4.10.1) | X<br>The PartCsyTransformation *with argument* command presumably has this effect. |
| sets the coordinate system with respect to which jog moves are made | SET_JOG_MATRIX (4.10.2) | X<br>The PartCsyTransformation *with argument* command presumably has this effect. |
| selects one of three coordinate systems as the current coordinate system | X | CoordSystem *with argument* (6.3.3.1) |
| gets the identity of the coordinate system currently in use | X | CoordSystem *no argument* (6.3.3.2) |
| sets the transformation giving the location of the part coordinate system | X | PartCsyTransformation *with argument* (6.3.3.4) |
| gets the transformation giving the location of the part coordinate system | X | PartCsyTransformation *no argument* (6.3.3.3) |

## 5.9 Settable Parameters

Settable parameters are parameters that may be set by a DME driver interface command. They are used by the low-level controller for rates, distances, time intervals, etc. used in its operation.

There is no point in being able to set a parameter if it is not used by the low-level controller. For interoperability, the meaning of each parameter and the use made of each parameter must be fully described in the interface specification.

Both documents are extremely inadequate in regard to settable parameters.

For most of the settable parameters, the *CMM-driver Specification* contains no description of what they mean and no reference is made to them outside of the description of commands that get and set them. The parameters for which this is the case include: approach_dist, measuring_force, moving_accel, probing_accel, retract_dist, and search_dist. In addition the syntax is unclear for moving_speed, moving_accel, probing_speed, and probing_accel. For all four of those, a phrase such as “percentage of maximum (0.7 is 70%)” is used, which might mean either “use the string ‘0.7’ and that will mean 70 percent”, or “use the string ‘70%’ and that will mean 0.7”.

As noted earlier, the *I++ DME-Interface* has omitted (apparently unintentionally) available commands for getting and setting parameters. The *I++ DME-Interface* does, however, provide a good description of how approach distance, search distance, and retract distance are used in executing the PtMeas command.

**Table 10: Settable Parameters**

| Normal Use of Parameter | Name in *CMM-driver Specification* | Name in *I++ DME-Interface* |
|---|---|---|
| distance from target point at which to start probing move | approach_dist<br>No semantics. | X<br>tool()->Approach() method mentioned, page 42, but no method defined. _Approach data member listed in 9.5.6. |
| distance by which sensor path may deviate from the programmed path during free space moves | fly_mode<br>Semantics fair. | X |
| force on probe at which a hit should be recorded | measuring_force<br>No semantics. | X |
| rate of acceleration for free space moves | moving_accel<br>No semantics, | Accel *for GoTo* (5.3 & 9.5.5) |
| speed at which to move in free space | moving_speed<br>Semantics weak. | Speed *for GoTo* (5.3 & 9.5.5) |
| rate of acceleration for probing moves | probing_accel<br>No semantics. | Accel *for PtMeas* (5.3 & 9.5.6) |
| speed at which to move while probing | probing_speed<br>Semantics weak. | Speed *for PtMeas* (5.3 & 9.5.6) |
| distance to back off after probe hit | retract_dist<br>No semantics. | X<br>tool().Retract() method mentioned, page 42, but no method defined. _Retract data member listed in 9.5.6. |
| distance past target point at which to stop probing move if no hit | search_dist<br>No semantics. | X<br>tool()->Search() method mentioned, page 42, but no method defined. _Search data member listed in 9.5.6. |
| time interval at which to report position | timed_pos_report<br>Semantics OK. | No system parameter, but is parameter of OnMoveReportE |
| axes whose position is set when a touch occurs | touch_report<br>Semantics OK. | No system parameter, but is parameter of OnMoveReportE |

## 5.10 Settable Parameter Commands

Settable parameter commands are for obtaining the values of parameters, obtaining the allowed values of parameters, and setting parameters. The parameters themselves are listed in the immediately preceding table.

The *CMM-driver Specification* is a little odd regarding command settable parameters. Eight of the eleven parameters can be set and obtained and their allowable values can be obtained. Probing_acceleration can be obtained and its allowable values can be obtained, but it cannot be set. Fly_mode and touch_report can be set but not obtained and their allowable values cannot be obtained. No explanation is given for this oddity, and it may simply be an oversight.

As noted earlier, the *I++ DME-Interface* has omitted (apparently unintentionally) available commands for getting and setting parameters, except in the case of parameters relating to the GoTo, EnableUser, and PtMeas commands.

**Table 11: Settable Parameter Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| gets the value of a named parameter | GET_PARAMETER (4.7) | X |
| gets the allowable values for a named parameter | GET_PARAMETER_INFO (4.8) | X |
| sets the value of a named parameter | SET_PARAMETER (4.9) | X |

## 5.11 Commands for Getting Configuration Data

Configuration items are pieces of information that can be obtained but not set by DME driver interface commands. This is information about the physical machine or the software being used or some aspect of the current mode of use.

The *CMM-driver Specification* provides one command, GET_CONFIG, to get configuration data. This command may take the name of any of twenty configuration items as an argument.

The *I++ DME-Interface* provides one command that gets one of the twenty items of configuration data specified in the *CMM-driver Specification*.

The *CMM-driver Specification* provides many more types of configuration data, most or all of which appear to be useful, so its provisions appears preferable.

**Table 12: Commands for Getting Configuration Data**

| Configuration Item | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| word size of controller<br>(8 bit, 16 bit, 32 bit, or other) | GET_CONFIG(CONTR_TYPE)<br>(4.13.1) | X |
| CNC capability of DME<br>(manual or CNC) | GET_CONFIG(CNC_TYPE)<br>(4.13.2) | X |
| control panel type | GET_CONFIGC(PANEL_TYPE)<br>(4.13.3) | X |
| double column mode<br>(on or off) | GET_CONFIG(DCOL_MODE)<br>(4.13.4) | X |
| double column number<br>(number of current column) | GET_CONFIG(DCOL_NUMBER)<br>(4.13.5) | X |
| type of head<br>(fixed, indexed, or servoed | GET_CONFIG(HEAD_TYPE)<br>(4.13.6) | X |
| CMM type | GET_CONFIG(MACHINE_TYPE)<br>(4.13.7) | Type<br>(6.3.2.10) |
| number of probe changers<br>(may be zero or positive) | GET_CONFIG(NUM_PRB_CHANGERS)<br>(4.13.8) | X |
| data for one probe changer | GET_CONFIG(PRB_CHANGER_DATA)<br>(4.13.9) | X |
| code number of product provider | GET_CONFIG (PROVIDER)<br>(4.13.10)<br>Only choices are B&S, LK, Zeiss. | X |

**Table 12: Commands for Getting Configuration Data**

| Configuration Item | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| revision number of product | GET_CONFIG (REV_NUMBER) (4.13.11) Does not specify what product the number is for when no product name is given. | X |
| whether the machine has a rotary table (0 or 1) | GET_CONFIG (ROTARY_TABLE) (4.13.12) | X |
| whether switching to scale coordinates is possible (0 or 1) | GET_CONFIG(SCALE_COORDINATES) (4.13.13) | X |
| current sensor type | GET_CONFIG(SENSOR_TYPE) (4.13.14) | X |
| whether temperature compensation is enabled (0 or 1) | GET_CONFIG(TEMP_COMP) (4.13.15) | X |
| whether the CMM being controlled is real or simulated | GET_CONFIG(USE_MODE) (4.13.16) | X |
| whether a direction vector is returned with touch points (0 or 1) | GET_CONFIG(TOUCH_VECTOR) (4.13.17) | X |
| total number of machine axes | GET_CONFIG(NUM_AXES) (4.13.18) | X |
| detailed information about one axis | GET_CONFIG(AXIS_DATA) (4.13.19) | X |
| order in which axis moves are made | GET_CONFIG(AXIS_SYNCH_GROUPS) (4.13.20) | X |

## 5.12 Tool and Tool Changer Data Commands

Tool data commands create or remove data about tips, sensors, and tool assemblies. These commands produce no motion of the tools. The *CMM-driver Specification* provides a large number of tool data commands. As mentioned earlier, the *I++ DME-Interface*, probably inadvertently, provides none.

The *CMM-driver Specification* provides three commands for dealing with tool changer data. The *I++ DME-Interface* provides none. Commands for actually changing tools, not just getting and setting data, are covered in section 5.7 of this report.

**Table 13: Tool Data Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| gets a list of the names of all available sensor assemblies | GET_ALL_ASSM (4.14.1) | X |
| gets a list of the names of all available tips in one named sensor assembly | GET_ALL_TIP (4.14.2) | X |
| gets a list of the names of all calibration positions of one named tip in one named sensor assembly | GET_ALL_SENSOR (4.14.3) | X |
| gets whether an assembly is analog, trigger, or hard | GET_ASSM_DATA (4.14.4) | X |
| gets data on one named tip in one named sensor assembly | GET_TIP_DATA (4.14.5) | X |
| gets data on one named tip and sensor in one named sensor assembly | SET_SENSOR_DATA (4.14.6) | X |
| gets data on a named sensor assembly | GET_TOOL_ASSM (4.14.7) | X |
| sets calibration data for an existing nominal sensor in a named sensor assembly for a named tip (thereby creating an actual sensor) | CREATE_SENSOR_ACTUAL (4.14.8) | X |
| sets data for a nominal sensor not already in a named sensor assembly for a named tip, and adds the nominal sensor to the assembly | CREATE_SENSOR_NOMINAL (4.14.9) | X |
| sets data for a tip not already in a named sensor assembly, and adds the tip to the assembly | CREATE_TIP (4.14.10) | X |
| sets data for a new sensor assembly | CREATE_ASSM (4.14.11) | X |
| deletes data for a named sensor assembly | DELETE_ASSM (4.14.12) | X |
| deletes a named tip from a named sensor assembly | DELETE_TIP (4.14.13) | X |
| deletes a named sensor from a named sensor assembly for a named tip | DELETE_SENSOR (4.14.14) | X |

**Table 14: Tool Changer Data Commands**

| What Command Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| gets data regarding the relationship between a tool changer and a tool assembly | GET_CHANGER_DATA (4.15.1) | X |
| adds data establishing a relationship between a tool changer and a tool assembly | CREATE_CHANGER (4.15.2) | X |
| deletes data regarding a relationship between a tool changer and a tool assembly, thereby removing the relationship | DELETE_CHANGER (4.15.3) | X |

## 5.13 Multiple Messages, Unsolicited Messages, and Error Messages

In both the *CMM-driver Specification* and the *I++ DME-Interface*, multiple messages may be sent by the DME driver in response to a DME driver command requiring an unspecified number of response messages. The *CMM-driver Specification* calls these "unsolicited" messages; the *I++ DME-Interface* does not call them unsolicited. The multiple shot commands of the *I++ DME-Interface* all lead to multiple responses.

Both the *CMM-driver Specification* and the *I++ DME-Interface*, provide that the DME driver may at any time send an error message. An error message may or may not identify a command. If the error cannot be identified with the execution of a command, the error message does not identify a command; both documents regard the error message as unsolicited in this case.

The *CMM-driver Specification* but not the *I++ DME-Interface* provides that unsolicited messages TOUCH, POSITION, and KEY may be sent "by the DME driver when teaching an inspection program" (and not under control of the application). The *CMM-driver Specification* does not explain how the application is informed that teaching is in progress.

Both documents identify errors that may occur and assign codes to them. The *CMM-driver Specification* provides a list of assigned codes. The *I++ DME-Interface* does not. Both documents need improvement in identifying all errors that may occur in executing each command. For many commands where errors may occur, the *CMM-driver Specification* does not list any possible errors.

In the following table, the *I++ DME-Interface* data response message (denoted by #) is listed as being equivalent to various specific *CMM-driver Specification* messages. This equivalence is only rough. The *I++ DME-Interface* is definitely the weaker of the two in regard to reporting data generated by a user on the DME driver side. It does not, for example, provide any explicit method of reporting which key has been hit. It is also not clear how to distinguish a user-generated touch point from a user-generated way point.

**Table 15: Multiple Messages and Error Messages**

| What Message Does | *CMM-driver Specification* | *I++ DME-Interface* |
|---|---|---|
| reports a touch point | TOUCH (5.2) | # (6.2.2.2 & 6.2.2.3) |
| reports current position while in manual mode | POSITION (5.3) | # (6.2.2.2 & 6.2.2.3) |
| reports the state of a control panel key pressed by the user | KEY (5.4) | # (6.2.2.2 & 6.2.2.3) |
| reports current position at fixed time and/or distance intervals | DRO_POS (5.5) | # (6.2.2.2 & 6.2.2.3) |
| reports that an error has occurred | ERROR (6.2) | ! (6.2.2.2 & 6.2.2.3) |