The NIST DMIS Interpreter
Version 2

Thomas R. Kramer
Frederick M. Proctor
William G. Rippey
Harry Scott

Intelligent Systems Division
National Institute of Standards and Technology
Technology Administration
U.S. Department of Commerce
Gaithersburg, Maryland 20899

NISTIR 6252
October 29, 1998

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

Acknowledgements

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

CONTENTS

FIGURES

TABLES

1 Introduction

The National Institute of Standards and Technology (NIST) DMIS interpreter is a software system that reads control code programs in the DMIS language (described below in Section 1.2), produces calls to a set of canonical commands for coordinate measuring machines, digests the results of taking measurements, and produces a file describing measured features and tolerances. The canonical command calls made by the interpreter can be used to drive a coordinate measuring machine. This report describes Version 2 of the DMIS interpreter (in this report “the interpreter”).

1.1 Background

1.1.1 Architecture Project

The NIST Manufacturing Engineering Laboratory (MEL) has conducted an architecture project for several years. Three MEL divisions have participated in the project. The primary objective of the project is to develop a reference model control architecture to support intelligent control systems for manufacturing. The architecture being developed is called the Intelligent Systems Architecture for Manufacturing (ISAM) [Albus].

1.1.2 Enhanced Machine Controller Project

The MEL Intelligent Systems Division (ISD) is carrying out an Enhanced Machine Controller (EMC) project. The primary objective of the EMC project is to build a testbed for evaluating application programming interface standards for open-architecture machine controllers. The EMC project has built several controllers. These are most often run in a research environment at NIST, but commercial installations of EMC controllers have also been done [Proctor].

1.1.3 Next Generation Inspection System Project

To advance the state of the art in inspection, ISD established the Next Generation Inspection System (NGIS) project. NGIS goals are (1) to maintain a next-generation inspection testbed for experimenting with open architecture controllers, interface standards, and multiple advanced sensors, and (2) to achieve fast, accurate, and flexible coordinate measurement of complex parts. A testbed has been assembled that consists of a coordinate measuring machine and advanced sensors, with a NIST Real Time Control System (RCS) architecture controller.

1.1.4 DMIS Interpreter First Version

As part of the architecture project, it was decided to put two levels of EMC controllers above the NGIS controller. The capability to interpret control programs was put in the control level immediately above the NGIS controller. DMIS was selected as the language for control programs.

There are several DMIS interpreters available commercially, but they connect directly to specific commercial machine tool controllers. The architecture project required an interpreter that could be driven by function calls from the controller in which it was resident, could drive lower control levels by an open programmatic interface, and could communicate with the environment and a feature-fitting mathematics module via open interfaces. The required interfaces are shown in Figure 1 on page 5. Such an interpreter is not commercially available, so it was decided to build the software at NIST.

A DMIS interpreter was built by the authors in 1996 and used in the control system for a coordinate measuring machine (CMM). A report on that first version of the interpreter [Kramer2] was published in April, 1997. All information from that report relevant to Version 2 of the interpreter is included in this report, so it is not necessary (or useful) to read that report in order to understand Version 2.

1.1.5 DMIS Interpreter Version 2

In mid-1997, research work at NIST developed the requirement that the DMIS interpreter be able to handle a series of small, related inspection programs. The DMIS language provides the capability to pass information from one program to a program run later. The capability is enabled by the existence of common variables in DMIS, but common variables had not been implemented in the first version of the interpreter. It was decided, therefore, to implement common variables. Study of how to do this indicated that a major revision would be required. It was decided to build Version 2 of the interpreter, and Version 2 was built during the middle of 1997. This report documents Version 2. Changes from the first version to Version 2 are discussed in Section 2.6, Appendix A.1.1, Appendix F.2.1, and Appendix F.3.1 of this report.

1.2 Overview of the DMIS Language

This section gives an overview of the DMIS language. Further details of the meaning of DMIS code are given in Section 3 of this report.

1.2.1 Introduction

DMIS (pronounced *DEE-miss* and standing for Dimensional Measuring Interface Standard) is a standard programming language for numerically controlled dimensional measuring equipment, primarily coordinate measuring machines (CMMs). Coordinate measuring machines from many manufacturers can be operated using programs written in DMIS.

DMIS was developed by the Consortium for Advanced Manufacturing - International. The most recent version of DMIS is Revision 3.0, which was completed in 1995 [CAM-I] and is ANSI American National Standard “ANSI/CAM-I 101-1995.” Both versions of the interpreter conform to Revision 3.0.

The DMIS specification [CAM-I] is large — 389 pages. It describes both an input language and an output language. The DMIS input language supports the following functions:

defining and measuring features (planes, circles, cylinders, lines, etc.)
defining and measuring tolerances
defining coordinate systems (and activating and deactivating them)
defining sensor characteristics and changing sensors
setting machine parameters (feed rates, probe tip radius, etc.)
machine motion - probing and free-space motion

The output language supports reporting the results of measuring features and tolerances and also serves as a log of input statements.

The general outline of a typical DMIS program is to define and measure some features on a part that serve to establish the coordinate system in which further measurements will be taken. Then, more features and tolerances on and among features are defined and measured in the newly established coordinate system. The measurements are analyzed, actual tolerances are calculated, and the results are saved in a file.

1.2.2 Statements, Lines, Major Words, Minor Words

DMIS is based on statements. A statement normally fits on a single line (a series of ASCII characters terminated by a carriage return and line feed). However, lines may be continued by putting the line continuation symbol (the $ character) as the last printable character on a line, so that a single statement may span several lines.

A typical statement consists of a major word, followed by a slash, followed by a mixture of minor words, labels, and numbers, for example `MEAS/PLANE, F(POCKET_BTM), 3`. Semantically, each statement represents a single command that is embodied in the major word. The minor words, the numbers, and the way in which the minor words and numbers are grouped specify parameters to the command and shades of meaning of the command.

The statement formats recognized by the interpreter are presented in a formal specification language in Appendix F. This includes about a quarter of the entire DMIS language, but it is the most heavily used quarter and covers perhaps 90 percent of what might be seen in typical inspection programs.

1.2.3 Programs

Statements may be collected in a file to make a program. A program consists of a DMISMN statement at the beginning[1], an ENDFIL statement at the end, and any number of other types of statements in between. The specification is not clear whether statements are intended to be usable outside of a program (as manual data input, for example). The interpreter requires an entire program.

1.2.4 Program Subunits

DMIS includes program subunits. A program subunit is a sequence of statements that forms a functional group. [CAM-I] defines ten types of program subunits. The interpreter implements only two of these types: measurement sequence and motion sequence. Each type of program subunit requires a particular type of first statement and a particular type of last statement.

A measurement sequence has a MEAS statement at the beginning and an ENDMES statement at the end. The function of a measurement sequence is to measure one feature. The significant statements inside a measurement sequence are PTMEAS statements, each of which is a command to measure a point.

A motion sequence has a GOTARG at the beginning and an ENDGO at the end. The function of a motion sequence is to move around in free space. Only GOTO statements may occur inside a motion sequence.

In the interpreter, we use the word “block” to mean either a statement or a program subunit.

1.2.5 Geometric Features

In DMIS, inspecting a part is done in terms of features and tolerances. Features in DMIS are mostly simple geometric elements. A complete list of DMIS feature types is: arc, <u>circle</u>, cone, cparln, <u>cylinder</u>, ellipse, gcurve, gsurf, <u>line</u>, object, parpln pattern, <u>plane</u>, <u>point</u>, rectangle, and sphere. The underlined five are implemented in the interpreter. DMIS features (such as the

1. By “DMISMN statement” we mean a statement using the major word DMISMN. In general, “XYZ statement” means a statement whose major word is XYZ.

cylindrical side of a hole) may be visible on a part being inspected or they may be purely conceptual (such as the line that is the axis of a cylindrical hole).

A DMIS program usually does not try to provide a complete description of the part to be inspected. Only those features that are to be measured or used indirectly for definitions need to be defined. There is no requirement on how much of the geometry of a feature must be present. For example, a line joining the centers of two circles is common in a DMIS program, even though there is no trace of it on the actual part.

DMIS does not provide a general geometric modeling capability. DMIS provides no capability to describe topology and no capability to perform modeling operations such as boolean subtraction of a feature from a part.

Each feature is considered to have both a nominal description, which is the one used when the feature is first defined, and an actual description, which is derived later on the basis of one or more measurements. The DMIS specification does not state whether a single nominal feature may correspond to more than one actual feature, but seems to assume that the correspondence is one-to-one.

Each nominal feature has a label that serves to identify it within a DMIS program. No other feature may share that label in the same program. The actual feature(s) corresponding to a nominal feature is (are) identified by the same label as the nominal feature.

1.2.6 Tolerances

DMIS tolerances also have labels that are unique among tolerances within a program.

Tolerances in DMIS do not belong to individual features. Tolerances are defined without reference to specific features and may be applied repeatedly. For example, a diameter tolerance of 0.1 mm might be defined and labelled DTOL1. Then a dozen circles might be tested to see if they meet DTOL1.

DMIS supports tolerances according to the ASME Y14.5-1994 Standard for Dimensioning and Tolerancing. Twenty-two types of tolerance are included. The interpreter implements seven of these to one degree or another: coordinate position, cylindricity, diameter, flatness, parallelism, perpendicularity, and relative position.

1.2.7 Comments

A DMIS program may include comments. A comment is a line that has two dollar signs as the first two characters. Such lines are to be ignored by the system executing DMIS statements. Comments may contain information useful to humans writing or using the program.

2 Overview of the Interpreter

2.1 Interpreter kernel

The inside of the interpreter is called the kernel in this report. Most of the software for the interpreter is part of the kernel. The kernel is accessible in a program only through the interfaces (which are not part of the kernel). The kernel will print error messages directly visible to the user, however.

2.2 Interpreter interfaces

The interpreter has five interfaces, as shown in Figure 1. Each interface is a collection of function calls; there is an application programming interface (API) for each such collection. Arrows show the direction of function calls. Return values (shown in parentheses) move in the reverse direction. cmm_do_something calls do not return anything, so nothing goes in the reverse direction.

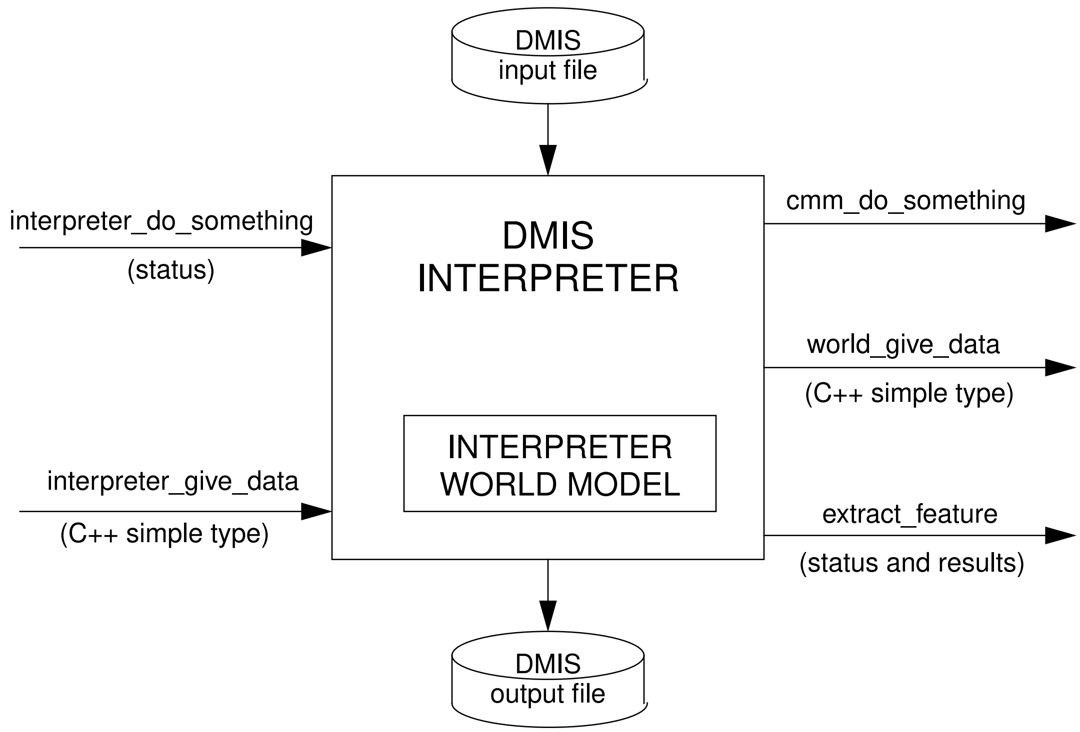

Figure 1. Interpreter Interfaces

The five APIs have been defined in the C++ programming language. For convenience, the function prototypes for all five APIs are given in a single header file. For each API there is a separate file that gives the definitions of the functions in the API. In this section we give only the names and arguments of the functions. More details are given in Appendix B, including a description of what each function does.

2.2.1 Telling the Interpreter What to Do

The functions in this interface (called interpreter_do_something in Figure 1) are:

```
interp_init()
interp_open_program (char * dmis_file_name)
interp_execute_next()
interp_close_program()
interp_exit()
```

2.2.2 Getting Data from the Interpreter

The functions in this interface (called interpreter_give_data in Figure 1) are:

interp_line()
interp_sensor_tip_diameter(char * sensor_name)

2.2.3 Telling the CMM What to Do

This interface is called cmm_do_something in Figure 1.

One of the main purposes of the interpreter is to tell the controller what the DMIS program says the equipment should do. To do the telling, a language is needed. A set of "CMM canonical commands" was developed to serve as that language. The CMM canonical commands are listed in Table 1.

The CMM canonical commands are atomic commands. Each command produces a single action.

The correspondence between executing a DMIS statement in the interpreter and the interpreter calling a CMM canonical function is usually one-to-one. Occasionally it is one-to-two. On the other hand, executing many DMIS statements (any statement that is a definition of a feature or tolerance, for example) requires no work on the part of a CMM. In such cases, the "advisory" CMM canonical function is called just to show that the interpreter did something. The "advisory" command contains a message but produces no CMM action. Without the "advisory" command, there would be many cases where the execution of a DMIS statement would result in no call to any CMM canonical command.

The canonical commands used in the interpreter were devised with three main objectives in mind. First, all the functionality of the existing NGIS had to be covered by the commands; for any function the NGIS can perform, there has to be a way to tell it to do that function. Second, it must be possible to interpret DMIS statements into canonical commands. Third, the canonical commands had to conform to the division of responsibility between the interpreter and the rest of the system, as described in Section 2.5.

Two sets of definitions for the CMM canonical functions have been written, and either set can be linked into the interpreter. The first set is used in the EMC controller for the NGIS testbed. Executing a function from this set causes a command message to be generated. When this command message is executed, the machine's actuators are activated. The second set is used in the stand-alone DMIS interpreter. Executing a function from the second set causes a line of text containing the command to be written to standard output or to a file.

```
ADVISORY(char * message)
ASSIGN_SENSOR_TO_SLOT(char * sensor_name, int slot_number)
CATCH_UP()
CHANGE_SENSOR(char * sensor_name)
DEFINE_SENSOR(char * sensor_name, double x_offset, double y_offset, double z_offset,
              double tip_diameter)
LOGGING_OFF()
LOGGING_ON(char * log_name)
MEASURE_POINT(double x, double y, double z, double i, double j, double k)
MESSAGE(char * text)
PROBE_RADIUS_COMPENSATION_OFF()
PROBE_RADIUS_COMPENSATION_ON()
PROGRAM_END()
PROGRAM_START(char * text)
ROTATE_TABLE(double position, CANON_DIRECTION wiseness)
SCAN_TO_POSE(double x, double y, double z, double i, double j, double k)
SET_COORDINATE_SYSTEM(double origin_x, double origin_y, double origin_z, double z_axis_x,
              double z_axis_y, double z_axis_z, double x_axis_x, double x_axis_y, double x_axis_z)
SET_DISTANCE_APPROACH(double distance)
SET_DISTANCE_CLRSRF(double distance)
SET_DISTANCE_DEPTH(double distance)
SET_DISTANCE_RETRACT(double distance)
SET_DISTANCE_SEARCH(double distance)
SET_FEED_RATE(double rate)
SET_PLANE(CANON_PLANE plane)
SET_ROTARY_RATE(double rate)
SET_ROTARY_ZERO(double angle)
SET_SCAN_DIST_INTERVAL(double dist_interval, CANON_AXIS axis)
SET_SCAN_INTERVAL_TYPE(CANON_INTERVAL_TYPE interval_type)
SET_SCAN_RATE(double rate)
SET_SCAN_TIME_INTERVAL(double time_interval)
SET_SCAN_TYPE(CANON_SCAN_TYPE the_type)
SET_TRAVERSE_RATE(double rate)
STRAIGHT_TRAVERSE(double x, double y, double z)
USE_ANGLE_UNITS(CANON_UNIT_ANGLE u)
USE_LENGTH_UNITS(CANON_UNIT_LENGTH u)
USE_TEMPERATURE_UNITS(CANON_UNIT_TEMPERATURE u)
```

Table 1. CMM Canonical Commands

Function arguments are written in ANSI C style. All functions return nothing.

2.2.4 Getting Data from the External World

This interface is called world_give_data in Figure 1.

The functions in this interface are called by the interpreter. These functions primarily obtain data that is collected during probing. The functions are:

```
CANON_MEASUREMENT_STATUS    MEASURE_POINT_STATUS()
double   CANON_PROBE_X()
double   CANON_PROBE_Y()
double   CANON_PROBE_Z()
double   CANON_CURRENT_X()
double   CANON_CURRENT_Y()
double   CANON_CURRENT_Z()
int      CANON_LOG_SIZE(char * log_name)
double   CANON_LOG_X(char * log_name, int n)
double   CANON_LOG_Y(char * log_name, int n)
double   CANON_LOG_Z(char * log_name, int n)
```

2.2.5 Extracting Feature Parameters from Arrays of Points

Each function in this interface takes an array of points and extracts parameters for a feature from it. The returned value is used only to indicate either OK or error.

The interpreter uses source code for these fitting functions provided by the NIST Algorithm Testing System (ATS) [Rosenfeld1, Rosenfeld2].

```
int extract_circle(double points [][3], int how_many, double tolerance,
    double * center_x, double * center_y, double * center_z,
    double * normal_i, double * normal_j, double * normal_k, double * diameter)

int extract_cylinder(double points [][3], int how_many, double tolerance,
    double * center_x, double * center_y, double * center_z,
    double * direction_i, double * direction_j, double * direction_k, double * diameter)

int extract_line(double points [][3], int how_many, double tolerance,
    double * point_x, double * point_y, double * point_z,
    double * direction_x, double * direction_y, double * direction_z)

int extract_plane(double points [][3], int how_many, double tolerance,
    double * point_x, double * point_y, double * point_z,
    double * normal_i, double * normal_j, double * normal_k)

int extract_point(double points [][3], int how_many, double tolerance,
    double * point_x, double * point_y, double * point_z)
```

2.3 Integrated or Stand-Alone Operation

The interpreter runs integrated with the EMC control system or as a stand-alone system. The program interfaces to the interpreter kernel are the same in the two cases. The interfaces seen by a user in the two cases are completely different. The stand-alone system provides a simple text-based command interface for the user; this interface is focused entirely on the interpreter. The EMC control system has a variety of textual and graphic interfaces, only a little of which deals with the interpreter.

In either case, the interpreter first reads the entire DMIS file and stores it as a data structure. The file reading includes a complete syntax check, as described in more detail in Appendix E. Then, the interpreter executes statements one at a time. If there is an error at any point, the interpreter sends a message identifying the nature of the error and stops running. If the error occurs during execution, execution stops at the statement where the error occurred, and it is not possible to restart the program from that point. To use a program that causes an interpreter error, the program must be edited to remove the error, and the program must be restarted at the beginning. Further details of error handling are given in Appendix E.1.

In both modes of use, if a DMIS output file is to be written (if there is a FILNAM statement in the input program) the interpreter always writes a DMIS output file named “output.dms”.

2.3.1 Stand-alone

The stand-alone mode is valuable because it allows a user to pre-test a DMIS program without having to run it on the machine controller. Any computer for which the stand-alone interpreter can be compiled can be used to pre-test DMIS programs. Pre-tests are conclusive tests of whether a program is interpretable or not because the interpreter runs exactly the same way in the stand-alone mode as it does integrated with the control system. Pre-tests do not show whether the program does what is intended, of course, and real-world data may defeat a program that runs happily in stand-alone mode using dummy data.

The architecture of the stand-alone interpreter is shown in Figure 2. A key feature of the stand-alone architecture is the dummy model of the external (outside the interpreter) world. This dummy model is essential because most DMIS programs require that data (primarily the location of measured points) be fed back into the program interpreter. To fill this need, the dummy model is changed by the cmm_do_something commands, and the world_give_data commands get data out of the dummy model. All dummy data is as close to nominal as possible. In the case of measuring points, for example, the dummy world modeler pretends that the location of the actual measured point is the same as the location of the nominal point.

In the stand-alone interpreter, the cmm_do_something functions each print a line of text (in addition to manipulating the dummy world model). The text goes to the user’s terminal by default, but may be redirected to a file. The main part of the line of text just echoes the function call. Each line also includes the sequence number of the function call (1, 2, 3, etc.) and the line number from the preprocessed DMIS file that gave rise to the function call.

The stand-alone interpreter has two modes of use: with or without a command interface.

With the command interface, the user has a finer level of control. It is started by giving the shell command `dmis`. This brings up a command interface that understands a handful of commands (a list of which is printed if the command `help` is entered). To interpret a DMIS program line-by-line, the user first gives an `interp_init()` command, then an `interp_open_program(input_file_name)` command (which causes the entire file to be read and an internal representation built), then a series of `interp_execute_next()` commands (each of which executes one statement from the program), then an `interp_close_program()` command. Either an `interp_exit()` or a `quit` command will quit the command interface. Also from the command interface, the user may give a `run_program(input_file_name)` command, which opens, executes, and closes the program.

In the second mode of use, the user gives a single command, in response to which the interpreter reads and interprets an entire DMIS file without bringing up any command interface. This mode is used by giving the shell command `dmis input_file_name`. In this mode, printed output from cmm_do_something function calls goes to the terminal by default but may be redirected to a file in the normal Unix manner, viz. dmis input_file_name > output_file_name. Even with output redirected this way, the DMIS output file output.dms is still written.

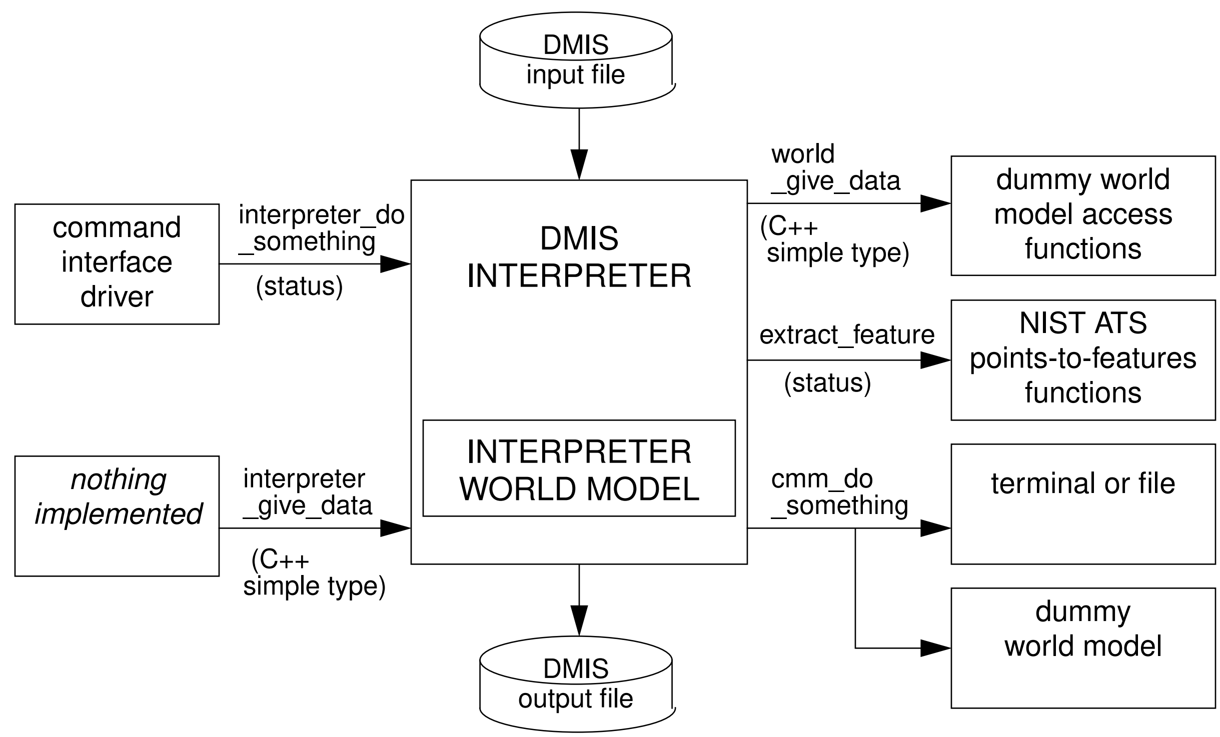

Figure 2. Stand-Alone Interpreter

2.3.2 Integrated with EMC Control System

The architecture of the part of the EMC control system that uses the interpreter is shown in Figure 3. The grey box is the EMC controller. The interpreter software is built into the controller.

In this integrated configuration, the control system tells the interpreter when to read the file and when to execute the next statement from the program.

The interpreter does not control machine actions directly. Rather, the interpreter calls CMM canonical commands that generate messages that are passed back to the control system, and the control system decides what to do with the messages.

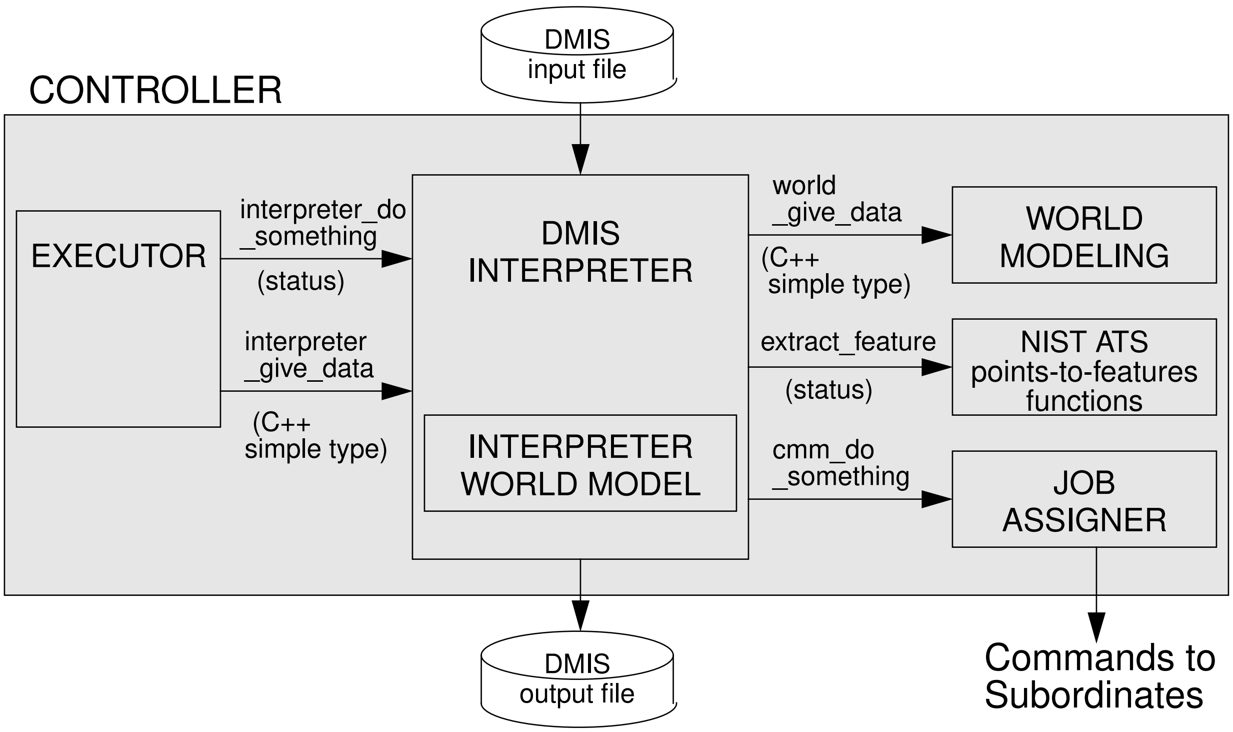

Figure 3. Interpreter Integrated in Controller

2.4 Major DMIS Interpreter Design Decisions

The following major design decisions were made regarding the interpreter.

The interpreter software runs in the same process as the executing system. This is to insure that the interpreter can be used conveniently and quickly. With the interpreter tightly integrated this way, communications with the interpreter consist simply of function calls and returned values. Without this tight integration, a more complex method of communicating with the interpreter would have been required. It would be feasible to implement the interfaces to the interpreter using messages sent and received through a communications system.

The executing system controls the interpreter and the CMM; the interpreter does not control the CMM directly. This does not show up in the interfaces to the interpreter. Rather, it is embodied in the definition of the cmm_do_something functions used in the integrated system. These functions just add things to do to the controller's queue. Thus, deciding when to do what is in the hands of the controller's job assigner, where it belongs.

Actions from cmm_do_something functions may be queued, but the interpreter may assume they are executed in order. The interpreter may direct that the queue be emptied before the interpreter is called again. This is so that the interpreter can maintain an accurate model of the world without having to make frequent calls to the world_give_data interface functions.

The executing system handles DMIS input and output via the interpreter. This is to keep the burden of dealing with features and tolerances centralized in the interpreter and off the rest of the executing system.

Enough of the DMIS language is implemented to meet the needs of the NGIS project and to handle DMIS programs for two specific parts (the test part for the material removal demonstration of the Department of Energy TEAM program, and the National Aerospace Standard 979 circle-diamond-square test part).

The interpreter should be easy to upgrade. In particular, it should be easy to add statements that deal with flow of control.

The interpreter must handle DMIS programs but not single DMIS statements entered by an operator. The capability to handle single statements could be added, if needed.

2.5 Division of Responsibilities

The DMIS interpreter is part of the executing system. This section discusses how responsibilities are divided between the DMIS interpreter and the rest of the executing system.

2.5.1 Control

The rest of the executing system performs control. The interpreter controls nothing; it only advises the rest of the system what the DMIS program says to do.

2.5.2 Languages

The interpreter understands DMIS statements and can interpret them to produce cmm_canonical commands. The rest of the executing system does not understand DMIS statements but can carry out cmm_canonical commands.

2.5.3 DMIS output

The interpreter produces DMIS output as required by DMIS programs. The rest of the executing system does not deal with DMIS output in any way.

2.5.4 Coordinate systems

The interpreter remembers all coordinate systems in a DMIS program. The rest of the executing system deals with one active coordinate system, which may be changed. One coordinate system both the interpreter and the rest of the system understand is the machine coordinate system. Each change of coordinate system is expressed to the rest of the system in terms of the machine coordinate system.

2.5.5 Features, Tolerances, and Variables

The interpreter remembers all nominal DMIS features (and corresponding actual features if and when actual features are created). The interpreter handles all DMIS tolerances. The interpreter handles all DMIS variables.

The rest of the executing system does not understand DMIS features, tolerances, or variables.

2.5.6 Units

The interpreter and the rest of the executing system both understand length (cm, inch, feet, m, mm), angle (decimal degrees, radians), and temperature (centigrade, Fahrenheit) units and can change them. It might be simpler to let the executing system deal with only one unit of each type;

this would certainly be feasible.

2.5.7 Sensors

The interpreter remembers all sensor definitions in a DMIS program, but data required by the rest of the executing system might not be in DMIS program. The rest of the system remembers sensors by name and can remember the diameter of a sensor tip. By using an interpreter_give_data command, the rest of the system can ask the interpreter for the tip diameter of a sensor by name. The rest of the executing system can change sensors.

2.6 Variables and Expressions

As mentioned earlier, the major changes between the first version and Version 2 of the DMIS interpreter were driven by the need to implement variables. This section discusses how this was done. Additional information about variables is given in Section 3.2.3.

In DMIS, all variables must be declared. This is done with the DECL statement [CAM-I, sec. 2.2.5, p. 5; sec. 3.1.2.2, p. 12; sec. 10.3.2, p. 272; sec. 10.3.3, pp. 273-274]. The interpreter only implements COMMON variables of type BOOL (boolean), INTGR (integer), or DOUBLE.

COMMON variables and their values persist between DMIS program runs, even if the system is powered down. Persistence of variables is achieved using a file named “DMIS_variables”. If a program uses variables, this file is read just after the program is read and is written at the end of program execution. The file lists the names of common variables, one per line, with the data type and value of each. It is an error if a variable declared in a program is not listed in the DMIS_variables file. The initial value of each program variable is set to the value given in the file. If the value of any variable is reset during a program, that value is given when the file is written at the end of the program. It is an error (and the interpreter will print an error message) if the DMIS_variables file does not exist and a program uses variables. A short sample DMIS_variables file is shown in Figure 4.

```
VAR3 double 88.77
AN_INT integer 707
INT2 integer 1234
BOO boolean true
```

Figure 4. Sample DMIS_variables File

A value may be assigned to a variable either by an ASSIGN statement or by an OBTAIN statement [CAM-I, sec. 10.5, pp. 284-287]. The manual provides for a third method not implemented in the interpreter. In the interpreter, only the parameters of nominal and actual circles, planes, and points may be used with OBTAIN.

A variable that has been given a value may be used in place of any parameter in any DMIS statement, so long as the variable is of the correct type. It is an error to use a variable in place of a parameter if the value of the variable has not been set.

The implementation of variables is made more difficult by the fact that variables should not be given values (except for those in the DMIS_variables file) or evaluated (even if included in the DMIS_variables file) when the program is read. It is necessary to wait until an explicit variable setting statement is executed to assign a value, and to wait until a DMIS statement referencing a variable is executed to evaluate the variable. Since feature and tolerance parameters may be variables, features and tolerances cannot be defined at read time; each is defined when a DMIS statement is executed. Any variable in a feature or tolerance definition is evaluated when the statement giving the definition is executed, and that value is used in the definition. If a variable changes value during a program, that does not affect any definition that has already been made.

The interpreter implements one tiny bit of expressions: parentheses. A number or a variable name may be enclosed in parentheses (or multiple nested sets of parentheses), and it will be recognized as an expression. The interpreter requires that the variable name be enclosed in at least one set of parentheses when a variable is used as a parameter value. In other words, variables are used only in expressions.

Because (1) the value of variables may change during a program, (2) values of parameters of features may be expressions containing variables, and (3) values of parameters of features may be referenced, it is necessary to have two sets of values for the parameters of each feature. One set (always called "exes" in the source code) contains the expressions and the other set contains the numerical values of the expressions at the time the feature definition is executed. The second set is the one that is used when the value of a feature parameter is referenced.

2.7 How the Interpreter Runs

To interpret a program, the interpreter is given an interp_init command, followed by an interp_open_program command, followed by many interp_execute_next commands.

The interpreter maintains a model (shown in Table 2) of the machine while it interprets. The interpreter uses the model in determining what cmm_canonical functions to call and what their arguments should be. The model is initialized when the interpreter is started by a call to the interp_init command.

When the interp_open_program command is given, the interpreter reads an entire DMIS program into active memory before any of the DMIS statements in it are interpreted. Then DMIS statements are interpreted one at a time.

In carrying out an interp_open_program command, the interpreter does the following:

1. The entire original DMIS program is read. It is stripped of comments and continued lines are joined. The DMIS program file "dmis_temp" is written.
2. The file "dmis_temp" is read, creating a large in-core structure, usable by the interpreter, that represents the entire DMIS program. This structure is made up mainly of substructures representing DMIS statements.
3. If the program contains variables, the file "DMIS_variables" is read.

Calling interp_execute_next causes the interpreter to interpret the statement that should be executed next. The statement that should be executed next is not necessarily the one on the line after the last line that was executed.

The structure of a DMIS program is, in general, a nested hierarchy of blocks of statements. The interpreter maintains a stack which mirrors the program structure. The stack is used to help decide which statement should be executed next and to remember important data about each block.

After a program has reached the last command to be executed (or in the middle of a program, if that is desired), an interp_close_program command should be given. This returns the interpreter to the state it was in before the program was opened.

Item	Meaning
double angle_factor	factor to multiply for radians
angle_unit_type angle_units	current angle units
datum_definition * current_system	current coordinate system
double current_position[3]	current position x,y,z
double current_table	current rotary table position
double defalt_feed	default linear feed rate
double defalt_rotate	default rotary table rate
double defalt_scan	default scan feed rate
double defalt_traverse	default linear traverse rate
datum_definition * default_system	default machine coord system
char error_message[TEXT_SIZE]	latest error message
int first_line	flag that first line found
double high_feed	high linear feed rate
double high_rotate	high rotary table rate
double high_scan	high scan feed rate
double high_traverse	ligh linear traverse rate
list_of_list_dmis_item item_stack	control structure
double length_factor	to convert to millimeters
line * exec_line	currently executing line
double low_feed	low linear feed rate
double low_rotate	low rotary table rate
double low_scan	low scan feed rate
double low_traverse	low linear traverse rate
double max_feed	maximum linear feed rate
double max_rotate	maximum rotary table rate
double max_scan	maximum scan feed rate
double max_traverse	maximum linear traverse rate
on_off_type mode_auto	whether in AUTO mode
on_off_type mode_man	whether in MAN mode
on_off_type mode_prog	whether in PROG mode
on_off_type output_dmis	whether to output DMIS
FILE * output_file	output FILE pointer
char output_file_name[TEXT_SIZE]	DMIS output file name
double points[1000][3]	array to hold point data
int point_number	number of points in array
on_off_type point_stuff_flag	whether to put point in array
double position_tolerance	positioning tolerance
double probe_x	last probed location x
double probe_y	last probed location y
double probe_z	last probed location z
char program_file_name[TEXT_SIZE]	program file name
list_dmis_item * program	program structure
on_off_type scan	whether to scan
on_off_type update_flag	whether to update positions

Table 2. Interpreter Internal Model

To stop the interpreter entirely, an interp_exit command should be given. This should not be done when a program is open.

2.8 Interpreter Model

The interpreter maintains three global variables: "_interp", "_interp_codes", and "_interp_limits". _interp is an instance of the "interp" class, whose attributes are shown in Table 2. All changeable data required by the interpreter (including the current DMIS program, for example) is incorporated in the _interp model. _interp_codes and _interp_limits are arrays of constants used in the interpreter.

2.9 Speed

The stand-alone interpreter, running on a SUN SPARCstation 20, read and executed a 720-line program in about 7 seconds. Just reading the same program file (not executing anything) took less than one second. Carrying out that program on a CMM using the integrated controller would take several minutes. Shorter programs take so little time for the stand-alone interpreter that it is hard to measure. We have not identified any situation in which the speed of the interpreter causes any problem in the rest of the system.

The only interpreter operation that we believe could take a significant amount of time is fitting features to sets of measured points.

2.10 Limitations of the Interpreter

The interpreter implements the parts of DMIS that are expected to be most heavily used, but this includes only about a quarter of the language. Several fairly common feature types, including sphere and cone, are not implemented. No statements for transfer of control, such as IF-ELSE, DO, or CASE are implemented. Expressions using operators and functions are not implemented.

In the first version of the interpreter, the functions that extract feature parameters from sets of points, which we obtained from another NIST division, did not appear to be reliable. New versions of these functions were received, incorporated in Version 2 of the interpreter, and tested. We believe they are reliable, so in Version 2, operation of those functions is *NOT* a limitation of the interpreter.

As mentioned earlier, the interpreter requires an entire program as input. The interpreter would need to be modified to handle single DMIS statements, a desirable capability for direct control by an operator.

3 Input

3.1 Overview

In general, allowable inputs are as described in [CAM-I] and discussed earlier in Section 1.2.

3.1.1 Case, White Space, Line Continuations, Comments.

The DMIS language is case insensitive [CAM-I, page 12]. Any letter may be in upper or lower case without changing the meaning of a statement, except that, within text strings, case is preserved. The interpreter implements these rules.

Blank lines are allowed in DMIS and in the input by the interpreter [CAM-I, page 11]. They are ignored. White space (spaces or tabs) is allowed between DMIS words, but not within them.

White space is preserved within text strings, however.

A single line may have a maximum length of 80 characters, but statements may consist of several lines by putting a '$' sign as the last printing character on each line to be continued. The maximum length of a statement allowed in [CAM-I, page 11] is 256 characters, but the interpreter will handle much longer statements. Line length up to 420 characters has been tested.

Lines are terminated with a carriage return (ASCII 13 (base ten)) and line feed (ASCII 10 (base ten)).

A line starting with '$$' is a comment and is ignored by the interpreter [CAM-I, page 11]. The double dollar sign is not allowed elsewhere. In the interpreter, it will cause an error during parsing if used anywhere except at the beginning of the line.

3.2 Input Statements

The formal specification of an allowable program is defined in Appendix F. The description here is intended to be consistent with the appendix. In order that the definition in the appendix not be unwieldy, some constraints imposed by the interpreter are omitted from that appendix. The list of error messages in Appendix E indicates all of the additional constraints.

3.2.1 Format of a DMIS Statement

A hierarchy of DMIS statement types is shown in Figure 5. Different statement types have different formats.

One large branch of the hierarchy is definitions. Definitions are any of the following. The term from Figure 5 is shown in parentheses.

coordinate system (datum_definition)
feature (feat)
label assignment (datdef)
rotary table (rotdef)
sensor (snsdef)
tool holder (thldef)
tolerance (tol)

Except for label assignment, a definition statement uses the form *thing defined = definition*. For example, a cylinder named CYL_A may be defined as follows.

```
F(CYL_A)=FEAT/CYLNDR,INNER,CART,0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 5.0
```

A DMIS statement may be a single DMIS major word on a line, such as:

```
ENDMES
```

More commonly, a statement is a DMIS major word followed by a slash and modifiers. Modifiers may be DMIS minor words, text strings, numbers, or labels of defined things. Here are a few examples of statements.

```
FILNAM/'teampart inspection results from demo1.dms'
FEDRAT/SCNVEL, MPM, 0.6
MEAS/POINT, F(START_PT), 1
PTMEAS/CART, 5.0, 0.0, -60.0, 0.0, 1.0, 0.0
```

3.2.2 Numbers

The manual [CAM-I, sec. 2.1.3, p. 5] is unclear in defining a valid number. In the formal specification used by the interpreter, given in Appendix F, the following rules regarding numbers may be found. In these rules a digit is a single character between 0 and 9.

A “LABEL_OR_INTEGER” is a sequence of one or more digits.

A “REAL” is a sequence of characters that does not qualify as a LABEL_OR_INTEGER but consists of (i) an optional plus or minus sign, followed by (ii) zero to many digits, followed, possibly, by (iii) one decimal point, followed by (iv) zero to many digits — provided that there is at least one digit somewhere in the number.

A “real” is a “REAL”[1] or a “LABEL_OR__INTEGER”. Character strings that form numbers are interpreted in the usual way. For example, a non-zero number with no sign as the first character is assumed to be positive.

With these definitions, the following observations are implicit.

Initial (before the decimal point and the first non-zero digit) and trailing (after the decimal point and the last non-zero digit) zeros are allowed but not required. A number written with initial or trailing zeros will have the same value when it is read as if the extra zeros were not there.

Numbers may have any number of digits, subject to the limitation on line length.

Exponential notation is not allowed.

3.2.3 Variables

The manual [CAM-I] states that a variable name may be one or more characters long, and the first character must be a letter. The manual puts two different upper limits on name length of 6 (on page 5) and 16 (on page 272). The interpreter uses the 6 limit. A variable may be given a value in several ways, as discussed in Section 2.6.

3.2.4 Line Number

The DMIS language does not provide for line or statement numbers. The interpreter keeps track of statement numbers by counting them. The first statement is assigned number 1, the second 2, and so on. In the original incoming DMIS file, statement numbers and line numbers may differ, because of blank lines, continued lines, and comments. In the preprocessed “dmis_temp” file, which has those things removed, statement numbers are the same as line numbers.

3.3 Words Recognized

The interpreter recognizes statements beginning with the 49 major word / minor word combinations shown in Table 3. These are all listed in the Index of Statements in [CAM-I, p. 389]. The meanings of the statements are as given in [CAM-I]. In most cases, not all of the possible variants of a statement are implemented. Any capability not explicitly included is excluded. Trying to use any excluded capability in a DMIS program will cause an error in the interpreter.

Three new major words have been added, as compared with the first version of the interpreter: ASSIGN, DECL, and OBTAIN, all of which deal with variables.

---

1. The formal specification is case sensitive, so “real” differs from “REAL”.

Code	Meaning
ASSIGN	assign a value to a variable
CONST Format 1	construct a feature
DATDEF	assign a datum label
DATSET	define a coordinate system
DECL	declare a variable
DISPLY	specify output devices and formats
DMISMN	begin a DMIS program
ENDFIL	end a DMIS program
ENDGO	end a GOTARG block
ENDMES	end a MEAS block
FEAT/CIRCLE	define a circle
FEAT/CYLNDR	define a cylinder
FEAT/LINE	define a straight line
FEAT/PLANE	define a plane
FEAT/POINT	define a point
FEDRAT	set a feedrate
FILNAM	assign an output file name
GOTARG	begin a block of free space moves
GOTO	make a free space move
MEAS	begin a block which measures a feature
MODE	select program modes
OBTAIN	set a variable value to a parameter value of a definition
OUTPUT	output data on features or tolerances
PRCOMP	turn probe tip diameter compensation on or off
PTMEAS	measure a point
RECALL	reactivate a coordinate system, sensor, or feature
ROTAB	rotate a rotary table
ROTATE	rotate a coordinate system
ROTDEF	define a rotary table
ROTSET	reset the angle of a rotary table
SAVE	save coordinate systems, features, etc. for later recall
SCNMOD	turn scanning mode on or off
SCNSET	specify how scanning will be done
SNSDEF Format1	define a sensor
SNSET	specify probing parameters
SNSLCT	change sensors
TEXT	send text to the operator or output file
THLDEF	define a tool holder
TOL/CORTOL	define a coordinate tolerance
TOL/CYLCTY	define a cylindricity tolerance
TOL/DIAM	define a diameter tolerance
TOL/FLAT	define a flatness tolerance
TOL/PARLEL	define a parallelism tolerance
TOL/PERP	define a perpendicularity tolerance
TOL/POS	define a position tolerance
TRANS	translate a coordinate system
UNITS	specify units
VFORM	specify output in vendor format
WKPLAN	select a working plane

Table 3. DMIS Words Implemented in the Interpreter

4 Conclusion

The main goals in creating Version 2 of the NIST DMIS interpreter were to implement variables and to make the interpreter independent of commercial software. These goals have been fully met. Version 2 is more powerful, much smaller (about 0.9 Mbytes vs 2.5 Mbytes), and slightly faster than was the first version. In addition, an improved set of feature-fitting functions is included in Version 2.

The interpreter is expected to be used at NIST as part of a system for controlling a coordinate measuring machine and/or a machining center (for probing) in one or more of four ISD projects: Architecture, Enhanced Machine Controller, Next Generation Inspection System, and Feature-Based Inspection and Control System.

Version 3 of the DMIS interpreter was built in September, 1998. Version 3 implements the same elements of the DMIS 3.0 standard as Version 2, but it is able to handle DMIS programs spread over several files, as well as single-file DMIS programs. The interface for telling the interpreter what to do has been modified in Version 3 to enable the new capability.

It is expected that the Version 2 software will be available, free of charge, on request.

For further information, contact the lead author of this document, Dr. Thomas Kramer, by email at kramer@cme.nist.gov (or thomas.kramer@nist.gov) or by telephone (301) 975-3518.

References

[Albus] Albus, J.S.; Meystel, A. M.; *A Reference Architecture for Design and Implementation of Intelligent Control in Large Complex Systems*; International Journal of Intelligent Control and Systems; Vol. 1, No. 1; 1996; 15-30

[CAM-I] Consortium for Advanced Manufacturing - International; *Dimensional Measuring Interface Standard*; Revision 3.0, ANSI/CAM-I 101-1995; CAM-I, Arlington, Texas; 1995

[ISO1] ISO 10303-11:1994; *Industrial automation systems and integration - Product data representation and exchange - Part 11: The EXPRESS Language Reference Manual*; ISO; Geneva, Switzerland; 1994

[ISO2] ISO 10303-21:1994; *Industrial automation systems and integration - Product data representation and exchange - Part 21: Clear Text Encoding of the Exchange Structure*; ISO; Geneva, Switzerland; 1994

[Kramer] Kramer, Thomas R.; Proctor, Frederick M.; *The NIST RS274/VGER Interpreter*; NISTIR 5754; National Institute of Standards and Technology; Gaithersburg, MD; 1995

[Kramer2] Kramer, Thomas R.; Proctor, Frederick M.; Rippey, William G.; Scott, Harry; *The NIST DMIS Interpreter*; NISTIR 6012; National Institute of Standards and Technology; Gaithersburg, MD; 1997

[Levine] Levine, John; Mason, Tony; Brown, Doug; *lex & yacc*; 2nd Edition; O'Reilly & Associates; Sebastopol, CA; 1992

[Loffredo1] Loffredo, David; et al; *STEP Utilities Reference Manual*; STEPTools Inc.; Troy, NY; 1994

[Loffredo2] Loffredo, David; et al; *ROSE Library Reference Manual*; STEPTools Inc.; Troy, NY; 1993

[Proctor] Proctor, Frederick M.; et al; *Simulation and Implementation of an Open Architecture Controller*; Proceedings of the SPIE International Symposium on Intelligent Systems and Advanced Manufacturing; Philadelphia, PA; 1995

[Rosenfeld1] Rosenfeld, David A.; *User's Guide for the Algorithm Testing System Version 2.0*; NISTIR 5674; National Institute of Standards and Technology; Gaithersburg, MD; 1995

[Rosenfeld2] Rosenfeld, David A.; *Reference Manual for the Algorithm Testing System Version 2.0*; NISTIR 5722; National Institute of Standards and Technology; Gaithersburg, MD; 1995

Appendix A Software Details

This appendix describes the software for the interpreter. The appendix is intended for users and programmers who want to modify the software or simply to understand it.

A.1 Overall Approach

The interpreter is written in C++. In Version 2, two additional languages are used: lex [Levine] and YACC (described lower on this page) [Levine].

A.1.1 Major Change from First Version

A major change was made in the overall software approach between the first version and Version 2 of the interpreter. The first version included the use of STEP[1] methods and tools; Version 2 does not. Specifically, in the first version, an EXPRESS information model of DMIS was built [ISO1], a second intermediate file was used (in STEP Part 21 format [ISO2]), and commercial software tools and libraries from STEPTools [Loffredo1], [Loffredo2] were used. The advantage of using STEP methods and tools was that the YACC actions were relatively simple to construct and access functions for dealing with the internal representation of the DMIS program were generated automatically. The disadvantages were that (1) the interpreter could be made available only to sites where STEPTools was available and (2) to change the software to handle additional words of DMIS, it was necessary to change the EXPRESS schema and reprocess it.

A.1.2 DMIS Object Classes and Access Functions

A tree of C++ classes is defined in the file interp.hh. The root of the tree is dmis_item. The leaf nodes of the tree are DMIS major words or major word - minor word combinations. The dmis_item tree is shown in Figure 5. A separate small tree of C++ classes is used to represent variables. Several class definitions not part of either tree are also given (along with much else) in the interp.hh file.

A set of C++ macros is used to simplify the definition of classes in the dmis_item tree. The macros define the same access functions as were used in the first version of the interpreter.

Lists are used extensively in the interpreter for data storage and access. A set of C++ macros for defining list classes is included in interp.hh.

A.1.3 YACC and lex

DMIS is a large language with hundreds of allowed formats for statements. It was clear that building a parser from scratch directly in C++ would be very time-consuming. YACC (Yet Another Compiler Compiler) grammar is a widely available, widely used language for specifying valid input for parsers. The DMIS manual is written in a style quite similar to that used for specifying a grammar in YACC. YACC, therefore, was chosen as the language for input parsing. Processing input specified in a YACC grammar requires a lexical scanner. The language for describing lexical input normally used with YACC is lex, so that was an obvious choice. The

1. STEP (Standard for the Exchange of Product Model Data), is a set of standards of the International Organization for Standardisation (ISO). All STEP standards are parts of ISO standard 10303. EXPRESS (not an acronym) is the official STEP information modeling language.

YACC grammar used in the interpreter is shown (without actions) in Appendix F.

DMIS input may include comments and line continuations. It is difficult to write a YACC grammar that deals with these items wherever they might appear, but it is rather easy to remove them as a pre-process. A brief lex specification was written for this purpose. Thus, the interpreter uses lex twice.

A file of C code for a lexical scanner and a file of C code for a parser are generated automatically by utilities that read the specifications and write the code. The usual utilities for handling lex and YACC files have the same names as the languages. However, there are other utilities for handling such files available from the Free Software Foundation, namely “flex” (fast lex) and “bison” (a mammal similar in appearance to a yak). These are somewhat superior to the original lex and yacc processors available to us and were used in building the interpreter.

As compared with the first version of the interpreter, the work done in the YACC parser is much greater in Version 2. In the first version, the return type of every YACC rule was a double; in Version 2, different rules have different return types. The actions taken by the rules are much more complex in Version 2.

A.1.4 Read First, Then Execute

For several reasons, it was decided to read and store the entire DMIS program before executing any of it. First, it saves time during execution, which may be important because of motion control requirements. Second, since a complete syntax check is performed during reading, it prevents getting part way through the execution of a program and then discovering it has an error. This often requires fixing the error and rerunning the entire program. Third, it is convenient to program. Input may be handled in one module and interpretation in another, rather than having to interleave them.

As a side effect of separating reading and execution, the functions in the kernel fall naturally into two parts: parsing functions and execution functions.

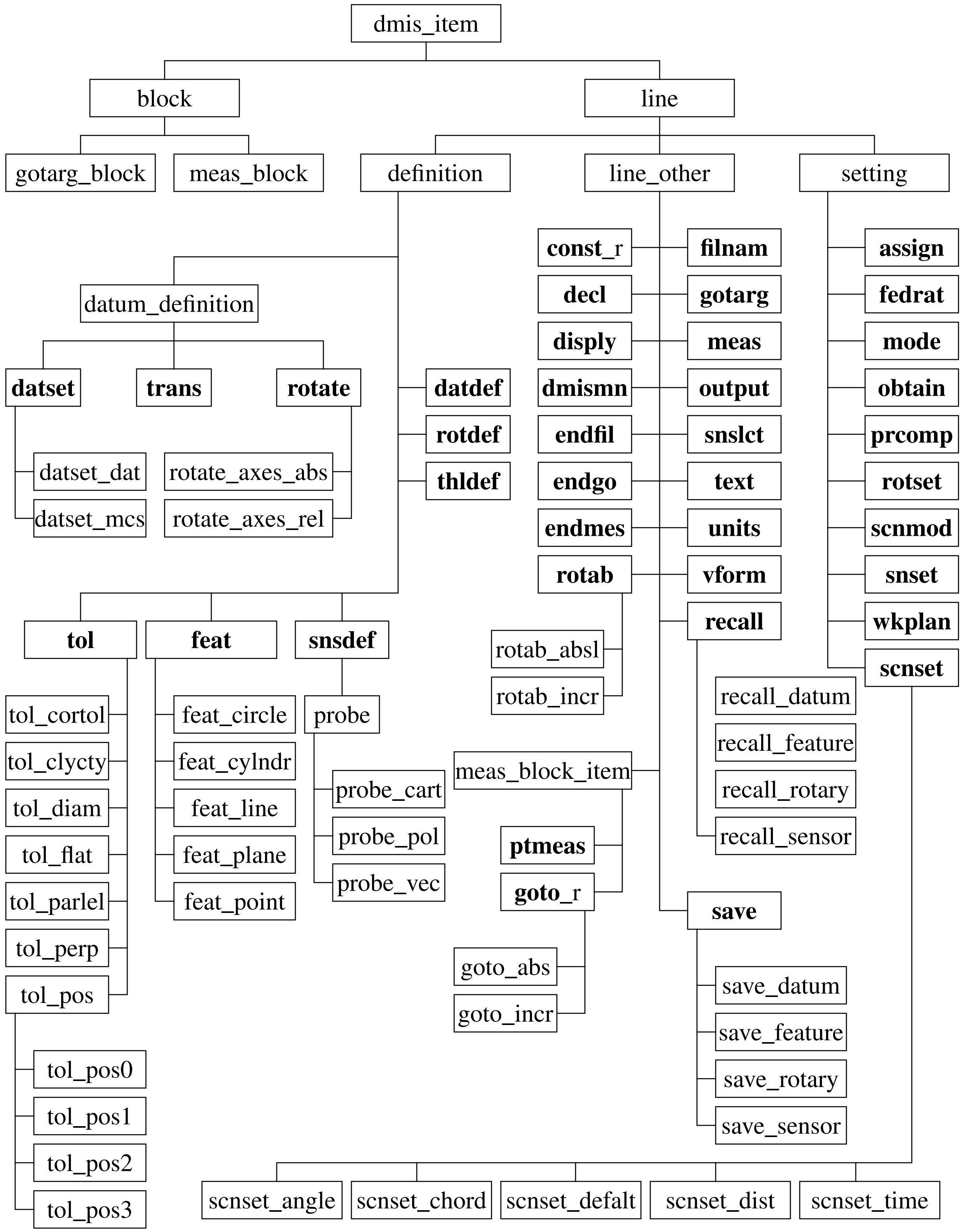

Figure 5. DMIS Class Hierarchy

DMIS major words are shown in boldface

A.2 Software Modules

Two methods of using the interpreter, stand-alone and integrated with the rest of EMC, are provided, as described in Section 2. The use of program files differs between the two methods. Some code is common to both, and some code differs.

A.2.1 Stand-Alone and Integrated

The program files used for both the stand-alone and integrated interpreter are:

C++ Header Files

1. cmm_canon.hh — 1,447 lines (mostly text)
   This is for the interpreter's five interfaces.
2. fit.hh — 364 lines
   This is for the NIST Algorithm Testing Service (ATS) feature fitting functions and was written by ATS personnel.
3. interp.hh — 2,323 lines
   This is for the interpreter kernel. It gives:
   a. various #includes and #defines of symbols
   b. several macros for defining classes hierarchically
   c. several enumerations
   d. the DMIS class hierarchy shown in Figure 5
   e. several macros for defining list classes
   f. the structure of the internal interpreter world model.
4. tk_common.hh — 35 lines
   This is a small collection of constants and macros to simplify programming.

C++ Function Definitions

1. cmm_canon_data_out.cc — 36 lines
   This is for the interpreter_give_data interface to the interpreter.
2. cmm_canon_extract1.cc — 5,399 lines
   This is the extract_feature functions. All but 511 lines of this was written by ATS personnel.
3. cmm_canon_interp.cc — 198 lines
   This is for the interpreter_do_something interface to the interpreter.
4. interp.cc — 19,100 lines
   This is the interpreter kernel. About a third of this was generated automatically by flex and bison from lex and YACC source, and half hand-written. The lex source totals 423 lines, while the YACC source is 960 lines.

Other

DMIS_variables - 24 lines
   This is a file that records DMIS common variables. On each line of the file are the name, type, and value of one common variable.

A.2.2 Stand-Alone Only

The program files used in the stand-alone interpreter only are:

C++ Header Files

1. rest_world.hh — 48 lines
   This defines the structure of the dummy world model used by the stand-alone interpreter.

C++ Function Definitions

1. cmm_canon_do_it.cc — 417 lines
   This is for the interpreter's cmm_do_something interface. The functions defined in this file print themselves. Some of them also alter the dummy world model to simulate being executed.
2. cmm_canon_data_in.cc — 152 lines
   This is for the interpreter's world_give_data interface. The functions extract data from the dummy world model.
3. driver.cc — 281 lines
   This provides an interface to the user.

A.2.3 Integrated Only

The interpreter program files used only in the integrated interpreter follow. In addition to these specific files, other files that are not considered to be part of the interpreter provide the world model actually used by the rest of the controller in which the interpreter lies.

C++ Function Definitions

1. cmm_canon_do_it.cc
   This is for the interpreter's cmm_do_something interface. The functions defined in this file mostly generate command messages.
2. cmm_canon_data_in.cc
   This is for the interpreter's world_give_data interface. The functions extract data from the actual world model.

A.3 Source Code Documentation

The source code is heavily documented. In general, for each function, four fields are given:

1. Returned Value - a description of possible returned values and the circumstances in which particular values may be returned. In most kernel functions, either OK or ERROR may be returned.

2. Side Effects - a description of the important side effects (things other than the returned value) of executing a function. Since the returned value of most functions is used to indicate error status, the side effects of most functions are important.

3. Called By - a list of functions that call the function being documented.

4. Argument Values - a one-line description of the meaning of each argument to a function, placed immediately after the declaration of the argument. This field is omitted if there are no arguments.

In addition to these four fields, most functions have a paragraph to a page of discussion. Where a function implements an algorithm for geometric or numerical calculation, the algorithm is described. Many citations to specific pages of the CAM-I manual are included in these discussions.

Appendix B Interpreter Interface Functions

As described in Section 2, the interface between a DMIS interpreter and a software system in which the interpreter is working comprises five sets of commands. The commands are represented here as C (or C++, which is identical for these usages) function definitions. The five sets are:

1. Functions that extract data from the interpreter.

2. Functions for the interpreter to call that extract data from the rest of the system.

3. Functions for the rest of the system to call to tell the interpreter to do something. These are normally called by a controller for which the interpreter works.

4. Functions for the interpreter to call that tell the rest of the system to do something which has been specified by a DMIS program or that the interpreter needs the rest of the system to do.

5. Functions for deriving “actual” feature parameters from point sets. For example, finding the center, plane, and radius of a circle, given three points in 3D space.

For some of these sets of functions, data type definitions are required. The definitions are given here as typedef’s at the beginning of each section.

B.1 Functions That Extract Data From the Interpreter

Functions that extract data from the interpreter are intended to be called by the system using the interpreter.

All the function names in this set start with “INTERP”

```
int INTERP_LINE()
```

This returns the line number of the line from the file dmis_temp (see Section 2.7) that is currently being executed by the interpreter. If no line is currently being executed, this returns the number of the line last executed. In the _interp model, the line number is updated when execution of a line starts.

```
double INTERP_SENSOR_TIP_DIAMETER(char * sensor_name)
```

This returns the diameter of the tip of the named sensor in current length units. If the interpreter has no record of a sensor of the given name, -1.0 is returned.

This would be used if the executing system does not know the diameter of the sensor tip and needs it to do probe tip radius compensation.

B.2 Functions for the Interpreter to Call to Get World Model Data

This set of functions is intended to be used by the interpreter. The value of data about the world outside the interpreter is expected to be returned in most cases.

```
typedef int CANON_MEASUREMENT_STATUS
#define CANON_OK 1
#define CANON_BAD 2
```

CANON_MEASUREMENT_STATUS is used as a return value of the measure_point_status function.

CANON_MEASUREMENT_STATUS MEASURE_POINT_STATUS()

MEASURE_POINT_STATUS returns the status of the last MEASURE_POINT action. OK means the last measurement worked successfully. BAD means it did not. Typically, a measurement will be bad if the probe did not trip, or it tripped before it should have, or it was not on, or it appeared to work but returned an out of range value, etc.

double CANON_PROBE_X()
double CANON_PROBE_Y()
double CANON_PROBE_Z()

CANON_PROBE_X returns the last recorded probe x-value (in the current coordinate system) resulting from a MEASURE_POINT function call. If probe compensation is on, this is the system's best estimate of the x-coordinate of the probed point. Otherwise, this is the system's best estimate of the x-coordinate of the location of the controlled point (normally the probe tip) when the point was contacted. CANON_PROBE_Y() and CANON_PROBE_Z() behave similarly.

double CANON_CURRENT_X()
double CANON_CURRENT_Y()
double CANON_CURRENT_Z()

CANON_CURRENT_X returns the current x-value of the controlled point (in the current coordinate system). CANON_CURRENT_Y() and CANON_CURRENT_Z() behave similarly.

int CANON_LOG_SIZE(char * log_name)

CANON_LOG_SIZE returns the number of points in the named log, if it exists. If the log does not exist, -1 should be returned.

double CANON_LOG_X(char * log_name, int n)
double CANON_LOG_Y(char * log_name, int n)
double CANON_LOG_Z(char * log_name, int n)

CANON_LOG_X returns the x-value of the nth point from the log of the given log_name. For the first point, n is 1. If this function is called and the log does not exist or n is larger than the number of points in the log, zero should be returned (changing this to return something more useful than zero would be a good idea, since zero could be a reasonable x-value). CANON_LOG_Y() and CANON_LOG_Z() behave similarly.

B.3 Functions to Tell the Interpreter What to Do

Conceptually, the interpreter has four states:

1. Down
2. Ready

3. Working (on a program)
4. Finished (working on a program)

These states are implicit and not explicit in the software.

Legal calls to functions in the interp-do-something interface, when in specific states, are shown in Table 4. The state in effect after each call (if the call does not return ERROR) is also shown in the table. Any call that has no place in the table to a function in the interp-do-something interface is an error. The interpreter is in the Down state initially.

Table 4. Interpreter State Transitions

In state	System May Call	Next state
Down	interp_init	Ready
Ready	interp_exit	Down
Ready	interp_open_program	Working
Working	interp_execute_next	Working (if call returns OK)
Working	interp_execute_next	Finished (if call returns EXIT)
Working	interp_close_program	Ready
Finished	interp_close_program	Ready

`int interp_close_program()`

This closes the current program. The return value is 0 (OK) or -1 (ERROR). It is an error to call this function if no program is open. It is OK to close a program even if it has not been executed to the end.

`int interp_execute_next()`

This causes the interpreter to decide which statement to execute next and to execute it. Execution of a statement always results in one or more calls to functions in the set of functions that tells the rest of the system what to do. The return value is 0 (OK), -1 (ERROR), or 1 (EXIT). EXIT is returned only when ENDFIL is interpreted. It is an error to call this function if a program is not open, or an earlier call to this function returned EXIT and the same program is still open.

`int interp_exit()`

This causes the interpreter to exit. This “undoes” interp_init. The return value is 0 (OK) or -1 (ERROR). It is an error to call this function if a program is open. It is OK to follow a call to interp_exit with a call to interp_init.

`int interp_init()`

This causes the interpreter to get ready to run. Once this function has been called and has returned OK, the interpreter stays initialized until a call to interp_exit. The return value is 0 (OK) or -1 (ERROR). It is an error to call this function if the interpreter has already been initialized.

`int interp_open_program(char * dmis_file_name)`

This causes the interpreter to open the named DMIS file, so that it is ready to be executed. The file stays open until a call to interp_close_program. The return value is 0 (OK) or -1 (ERROR). If the file cannot be opened (if it does not exist, for example), ERROR is returned. It is an error to call this function if a file is already open or if the interpreter has not been initialized.

B.4 Functions to Tell the Rest of the System What to Do.

B.4.1 Discussion and Issues

Functions that tell the rest of the system what to do are intended to be atomic. In other words, each function does one thing only. DMIS statements are not all atomic. Thus, executing one DMIS statement will often result in more than one of these functions being called.

A recurring issue for canonical functions is: For a given functionality, should the interpreter handle it internally or should the interpreter assume the system which executes canonical functions will handle it? A typical example is length units (suppose the alternatives in the input are inches and millimeters). If the executing system always expects millimeters, the interpreter should turn on an internal converter when it reads a DMIS statement that says to use inches. Thereafter, all lengths should be converted to millimeters by the interpreter before being used in canonical commands. If the executing system can handle different units itself, then there are two choices: either the interpreter can do the conversion or the interpreter can make the function call: USE_LENGTH_UNITS(CANON_UNIT_INCH); that function is not even defined in the previous case. One criterion for deciding where to put functionality is that most commercially available CMM controllers should have the functionality if a canonical command can call for it.

A compromise position could be taken where the functionality is shared between the interpreter and the CMM. For example, we might allow only two canonical length units (inches and millimeters), convert feet to inches in the interpreter, and convert centimeters and meters to millimeters in the interpreter. Currently, there is no sharing; the interpreter assumes the CMM can handle all five length units.

A closely related problem is what to do when DMIS allows symbolic values for items that are normally numeric. For example, DMIS allows HIGH, LOW, and DEFALT for feed rates. The interpreter will convert such symbolic values to numeric values by referring to configuration data, which is read in at initialization time. If the CMM may or must be given a symbolic value, the controller which gives commands to the CMM can convert a numeric value back to a symbolic value when the canonical command is converted to the CMM's native language. That controller can get the right symbolic value by referring to the same configuration file.

A DMIS program may define many instances of DMIS types such as features, sensors, and tolerances. A major issue is what view should the executing system have of these things.

SENSORS - Since the executing system must manipulate the sensors, we assume here that the system can identify sensors by name and can remember any number of sensor names.

FEATURES - We assume the executing system knows nothing about the feature types defined in DMIS.

TOLERANCES - like features

DATUMS - like features

Output - The view of the DMIS 3.0 specification [CAM-I, p.332] is that several methods of information output may be attached to a system running a DMIS program (video display, magnetic storage (disk file), paper printer, or communications port), and the output format may be either DMIS format or a vendor format. It is assumed in these canonical functions that the DMIS interpreter will handle all output in DMIS format (since the interpreter is supposed to be the DMIS expert). Thus, there should never be any canonical functions for producing DMIS output. This version of the canonical functions does not deal with vendor format output, either. Future versions of these canonical commands should deal with vendor format output.

This set includes some functions (currently only CATCH_UP) that incorporate a view of how the rest of the system works.

B.4.2 Types

```
typedef int CANON_PLANE
#define CANON_PLANE_XY 1
#define CANON_PLANE_YZ 2
#define CANON_PLANE_XZ 3
```

[CAM-I, p. 122] allows these planes. CANON_PLANE is used in the SET_PLANE function to identify the plane to use.

```
typedef int CANON_UNIT_ANGLE
#define CANON_UNIT_ANGDEC 1
#define CANON_UNIT_ANGDMS 2
#define CANON_UNIT_ANGRAD 3
```

[CAM-I, p. 180] allows these angle units. ANGDEC is angle in degrees with a decimal, e.g. 34.0779. ANGDMS is angle in degrees, minutes, and seconds (all integer), e.g. 4:03:47, but this form is currently not supported by the interpreter. ANGRAD is angle in radians, e.g. 4.1976. CANON_UNIT_ANGLE is used in the USE_ANGLE_UNITS function.

```
typedef int CANON_UNIT_LENGTH
#define CANON_UNIT_CM 1
#define CANON_UNIT_FEET 2
#define CANON_UNIT_INCH 3
#define CANON_UNIT_M 4
#define CANON_UNIT_MM 5
```

[CAM-I, p. 180] allows these length units. CM is centimeters, FEET is feet, INCH is inches, M is meters, MM is millimeters. CANON_UNIT_LENGTH is used in the USE_LENGTH_UNITS function.

```
typedef int CANON_UNIT_TEMPERATURE
#define CANON_UNIT_TEMPC 1
#define CANON_UNIT_TEMPF 2
```

TEMPC is degrees centigrade. TEMPF is degrees Fahrenheit. [CAM-I, p. 180] allows these temperature units. CANON_UNIT_TEMPERATURE is used in the USE_TEMPERATURE_UNITS function.

```
typedef int CANON_DIRECTION
#define CANON_CLOCKWISE 1
#define CANON_COUNTERCLOCKWISE 2
```

CANON_DIRECTION is for the direction of rotation of a rotary table. It is used in the ROTATE_TABLE function.

```
typedef int CANON_SCAN_TYPE
#define CANON_DRAG   1
#define CANON_NONCON 2
#define CANON_PECK   3
```

CANON_DRAG means a contact probe is dragged along the surface. CANON_NONCON means the probe is moved near the surface but not touching it (non-contact). CANON_PECK means the probe is primarily moved near the surface but not touching it and from time to time stops moving along the surface and moves to touch the surface and retract. [CAM-I, p. 245] allows these scan types. CANON_SCAN_TYPE is used in the SET_SCAN_TYPE function.

```
typedef int CANON_INTERVAL_TYPE
#define CANON_DIST 1
#define CANON_TIME 2
```

DIST is distance in current units. TIME is time in seconds. [CAM-I, p. 245] allows these scan interval types. CANON_INTERVAL_TYPE is used in the SET_SCAN_INTERVAL_TYPE function.

```
typedef int CANON_AXIS
#define CANON_AXIS_X 1
#define CANON_AXIS_Y 2
#define CANON_AXIS_Z 3
#define CANON_AXIS_NONE 4
```

These are the axis types needed for referring to axes. CANON_AXIS is used in the SET_SCAN_DIST_INTERVAL function.

B.4.3 Functions

`void ADVISORY(char * message)`

This indicates the interpreter has changed something internally in the course of executing a DMIS statement. The CMM controller should do nothing when this function is called. It is provided only to show that a statement has been executed and to allow a human-interpretable description of what the interpreter did.

`void ASSIGN_SENSOR_TO_SLOT`
`(char * sensor_name, int slot_number)`

This tells the executing system that the named sensor is in the numbered changer slot. The executing system is not required to do anything in response to this command (not even remember the correspondence, although it may do that). This command is for executing THLDEF [CAM-I, p. 166]. See discussion of CHANGE_SENSOR below.

`void CATCH_UP()`

The interpreter expects that the actions specified by all function calls in this section will be carried out as described in the order in which the calls are made (or possibly in parallel if the effects are the same). The interpreter does not usually know when the actions are carried out. In many cases the interpreter needs data collected as a result of the actions, so the interpreter needs to know that the actions have been carried out. The CATCH_UP function is provided to deal with this situation. After the CATCH_UP function is called by the interpreter, the rest of the system should not call the interpreter again until all previous actions specified in canonical function calls have been executed.

Currently, the interpreter is not trying to check that a CATCH_UP has been carried out. It might be useful to have the rest of the system notify the interpreter that it is caught up.

If the operation of the rest of the system is such that actions are carried out immediately after a function call is made (before doing anything else), which is one standard mode of operation, executing CATCH_UP is a null operation; the system is always caught up. If the rest of the system is queuing actions, however, the queue should be emptied before the interpreter is called again.

On the next call by the executing system to interp_execute_next, following a call by the interpreter to CATCH_UP, the interpreter will call one or more functions asking for data before making any calls to the functions in this section.

The functionality of CATCH_UP could be obtained, alternatively, by having the interpreter return a value that means catch up from a call to one of the functions in the interp_do_something interface. This is the way it was implemented in EMC machining center controllers. Making CATCH_UP a function is just a higher-profile method.

`void CHANGE_SENSOR(char * sensor_name)`

It is assumed that only one sensor is used by a CMM at a time. This changes the sensor to the one named in function call. If the named sensor is already being used, that is OK and nothing need be done.

The sensor_name argument must be the name of a sensor previously defined with SNSDEF. Since the interpreter maintains information about sensors defined with SNSDEF, it is anticipated that the rest of the system may make canonical function calls to the interpreter to get this information.

Changing the sensor should also automatically change system variables that depend upon the definition of the sensor, such as the current location.

Alternatively, tool holder slot number might be used as an argument to CHANGE_SENSOR, and each sensor would be identified with a slot number. If the CMM did not have slots, the sensors could still be numbered. DMIS includes the THLDEF (tool holder definition) statement to support the use of numbered slots. It would be feasible to require that THLDEF be used before a SNSLCT statement referring to that sensor. However, existing DMIS programs (teampart.dmis, in particular) do not always use THLDEF. Some systems are smart enough to keep track of sensors (or other objects) without having to put them in the same place all the time. This can save a lot time in tool changes. Forcing such systems to use slot numbers would work against this desirable feature.

Another alternative would be to have both slot number and name as arguments to CHANGE_SENSOR. If the slot number were non-negative, it would be intended to be a real slot number. It it were negative, that would indicate that it is not a valid slot number and should be ignored. The name would be either an empty string or the correct name when a non-negative slot number was used, and would be the correct name if a negative slot number was used. One or both of the slot number and name would have to be correct.

An additional consideration is that the assignment of sensors to slots may change even when the inspection program does not change. If sensors are identified by name only in a program, changing slots is not a problem. The executing system may have its own data about the location of sensors, which is not included in the program. This is common practice in the closely related field of machining. If the interpreter needs this information, it can be downloaded to the interpreter at initialization time (what the NIST RS274 interpreters do [Kramer]), or the interpreter might ask for it when it is needed (not implemented).

void DEFINE_SENSOR(char * sensor_name, double x_offset, double y_offset, double z_offset, double tip_diameter)

The x, y, and z_offsets of this function locate the center of the tip of the named probe with respect to the coordinate system of the probe. It also provides the diameter of the probe tip, which is assumed to be spherical.

The coordinate system of the probe is expected to be known.

It is suggested, but not required, that the coordinate system of the probe be such that, when the probe is attached to a CMM in a normal manner, the origin of the probe coordinate system should be at the mount location of the CMM, and the axes of the probe coordinate system should be parallel to the corresponding axes of the CMM. This location will simplify the transformations needed for deriving CMM axis positions from desired probe positions.

`void LOGGING_OFF()`

This turns logging point data OFF.

`void LOGGING_ON(char * log_name)`

This turns logging point data ON. If SCAN_TO_POSE is called when logging point data is ON, the data points taken during the scan are saved under the name log_name (normally a file name but not required to be a file). If there are already some points saved under that name, they are preserved, and the new points are added after the old ones. It is an error to call LOGGING_ON twice without an intervening LOGGING_OFF.

`void MEASURE_POINT`
`    (double x, double y, double z, double i, double j, double k)`

This is called for executing the DMIS PTMEAS statement [CAM-I, p. 208] when SCNMOD is OFF.

All the parameters to this command and the points and vectors defined below refer to the currently active coordinate system.

The following should happen when this function is called. These actions are supposed to be what is intended by the DMIS spec. Because the spec is vague, however, it is hard to be sure this is what the spec intends. It may be that the spec is vague intentionally, to allow for various implementations.

SETUP

Let C be the location of the controlled point (center of probe tip) before this instruction is executed.

Let P = (x,y,z) be the nominal location of the point to be probed. P must lie on a surface, but the system is not required to check this.

Let V = (i,j,k) represent a vector. If the surface has a normal at P, V should point in that direction, but no verification of this is required. If the surface normal is not defined (the tip of a cone, for example) V should point away from surface at P (preferably into a region where a normal to the surface in the vicinity of P cannot be drawn, but this is not required). If V is not a unit vector (within the system's tolerance for unit vectors), probe status is set to CANON_BAD, and execution is finished (doesn't even start, actually).

Let A be a point lying in the direction of V, one approach distance (as given with SNSET [CAM-I, p. 150] or by default) from P.

Let T be the location of the controlled point when the probe trips, if it trips during execution of this command.

Let S be a point lying in the direction opposite V one search distance (as given with SNSET or by default) from P.

ACTION

1. Move the controlled point in a straight line from C to A at the given feed rate (whatever the

system can do to control acceleration and deceleration is assumed to be adequate). If the probe trips during this move, probe status is set to CANON_BAD, and execution is finished. It may be desirable to allow traverse rate here because it is common for lots of points to be probed one after the other, and using traverse rate could save a lot of time. On the other hand, using the feed rate may help prevent broken probes, since it may be possible to stop quickly enough to avoid breaking a tripped probe if it is moving at feed rate where it would not be possible if it were moving at traverse rate.

2. Move the controlled point in a straight line from A towards S at feed rate.

Stopping at the intersection of lines CA and AS (point A) is optional.

If the probe trips during this move before it is within the system's tolerance for feed rate or direction, probe status is set to CANON_BAD, and execution is finished.

If the probe does not trip by the time it reaches S, it is stopped at maximum normal deceleration and moved back to S, probe status is set to CANON_BAD, and execution is finished.

3. Otherwise, if the probe trips during the move from A towards S, it is stopped at maximum normal deceleration and moved at feed rate to T (might change to traverse rate).

4. Without stopping, the probe is moved at feed rate (might change to traverse rate) in the direction of V one retract distance (as given with SNSET or by default) from T. If the probe trips during this move, probe status is set to CANON_BAD, and execution is finished. Otherwise, probe status is set to OK.

After this command has been executed OK, the executing system must have recorded the following - which must be available for reading out at least until the next command that moves the probe is executed.

1. the coordinates of the final position of the probe.

2. The coordinates of a point. The manual [CAM-I, p. 177] implies strongly that:

a. If probe compensation is on (as determined by the PRCOMP DMIS statement), this should be the system's best estimate of the actual coordinates of P.

b. If probe compensation is off, this should be the coordinates of T.

The approach, retract, and search distances could be parameters to the MEASURE_POINT command, rather than being assumed available within the system.

If the probe is not a touch trigger probe, a lot of the above does not make sense. The meaning of this command for other than a touch trigger probe should be rethought.

`void MESSAGE(char * text)`

This indicates that a message is to be displayed for the operator. The nature of the display is up to the receiving system, but will normally be on a computer monitor that is visible to the operator.

`void PROBE_RADIUS_COMPENSATION_OFF()`

`void PROBE_RADIUS_COMPENSATION_ON()`

The model of a probe tip (assumed both in the canonical commands and in DMIS [CAM-I, p. 126]) is that the tip of a touch trigger probe is a sphere, cylinder, or disk with its center at a known distance and direction from a point at the base of the probe. The diameter of the sphere, cylinder, or disk is given with the SNSDEF statement. In the case of a cylinder or disk, the orientation of the axis is also known.

These commands turn probe radius compensation off and on for a touch trigger probe. We assume that the controlled point is the center of the probe tip.

When probe radius compensation is off and a probing is made, the location of the center of the probe tip at the time the probe is tripped should be reported. The manual [CAM-I, p. 177] refers to this as “raw data.” Raw data is not defined in the glossary of the manual.

When probe radius compensation is on, the system’s best estimate of the location of the contact point that caused the trip should be reported.

`void PROGRAM_END()`

A call to PROGRAM_END signals the end of a program. After this function has been called, the only function from the cmm_do_something interface that is valid to call is PROGRAM_START. The executing system is not required to do anything to carry out the PROGRAM_END function, but may perform termination actions if needed.

`void PROGRAM_START(char * text)`

PROGRAM_START carries out the DMISMN statement [CAM-I, p. 170], which must be the first statement of a DMIS program.

A call to this function signals the beginning of a program. It is an error to call this function two times without an intervening call to PROGRAM_END. The executing system does not have to do anything to carry out this function, but it is likely that some sort of initialization will be carried out in most executing systems.

The text supplied with DMISMN is arbitrary. It is passed on by the PROGRAM_START function in case the executing system wants to do something with it.

```
void ROTATE_TABLE
    (double position, CANON_DIRECTION wiseness)
```

ROTATE_TABLE assumes that there is one rotary table, and it may be turned to a given position, which is assumed to be in current angle units (degrees or radians) from a home position. Position values range 0 <= angle < 360 for degrees and 0 <= angle < 2Pi for radians. Values of position outside this range are illegal. The wiseness must be CANON_CLOCKWISE or CANON_COUNTERCLOCKWISE.

Wiseness is as viewed from the side of the table on which the workpiece is usually fixtured.

The motion should be a smooth acceleration from the start position to the programmed rotational velocity and a smooth deceleration to a stop at the end position.

The DMIS language allows for any number of rotary tables, but these cmm_canonical functions allow for only one.

A command to rotate to the current nominal position should result in one full revolution of the table in the stated direction. The executing system must keep track of current nominal position, as given by ROTATE_TABLE commands.

```
void SCAN_TO_POSE
    (double x, double y, double z, double i, double j, double k)
```

This is called for executing the DMIS PTMEAS statement [CAM-I, p. 208], when SCNMOD is ON.

This command is intended to be implementable for touch trigger probes doing peck probing, for position probes which can be dragged along a surface, or for non-contact probes which might be moved along near a surface (either servoed to stay near the surface or not).

All the parameters to this command and the points and vectors defined below refer to the currently active coordinate system.

The following should happen when this function is called. These actions are supposed to be what is intended by the DMIS spec, but the spec is vague, so it is hard to be sure this is what the spec intends.

SETUP

Let C = (Cx, Cy, Cz) be the location of the controlled point (center of probe tip) before this instruction is executed.

Let P = (Px, Py, Pz) be the nominal location of the goal point (given by x, y, and z in the function prototype above).

Vector (i, j, k) represents a unit vector pointing in the intended direction of the probe at the end of the move. If that is not a unit vector (within the system's tolerance for unit vectors), probe status is set to CANON_BAD, and execution is finished (does not even start, actually).

ACTION

Move the controlled point from C towards P at the current scan feed rate, taking measurements periodically. Measurements are taken either at fixed time intervals or fixed intervals of distance along the line, according to whether TIME or DIST was used in the last call to SCNSET.

The motion from C to P should be "straight" in some sense, but is not necessarily a straight line. Exactly what this means is not defined, but the following paragraph gives an example of one type of motion that is intended to be permitted.

Example: A non-contact probe is moved so that the probe tip stays in a plane which contains the line from C to P and is parallel to the Z-axis. During this move, the Z location of the probe tip is controlled so that the it stays near the surface of the part being measured but does not hit the part.

If some problem is detected during this move, (e.g., motion control fails or the probe breaks off) probe status is set to CANON_BAD, and execution is finished. Otherwise, probe status is set to CANON_OK.

The motion stops when the probe has “reached the goal pose.” In other words, position and orientation (or selected components of them) are within some implementation-dependent tolerance zone of the goal pose (or selected components of it).

AFTEREFFECTS

After this command has been executed OK, the executing system must have recorded the coordinates of the final position of the probe, which must be available for reading out at least until the next command that moves the probe is executed.

If data logging is ON, after this command has been executed OK, the executing system must have recorded the coordinates of a number of points in a file of the given name (or using any other method of storing the data so it can be retrieved by name). The file must be available for reading out at any time until a PROGRAM_END command is received. The executing system must also remember how many points there are in the file. The points in the file should be points on the feature being measured. If data logging is OFF, no point data (other than the final position) need be saved.

```
void SET_COORDINATE_SYSTEM(
    double origin_x, double origin_y, double origin_z,
    double z_axis_i, double z_axis_j, double z_axis_k,
    double x_axis_i, double x_axis_j, double x_axis_k)
```

The arguments to this function are in terms of a machine’s default coordinate system (the location of which relative to the machine hardware is known).

The arguments describe a coordinate system with its origin at (origin_x, origin_y, origin_z), its Z axis pointing in the direction of the vector (z_axis_i, z_axis_j, z_axis_k), and its X axis pointing in the direction of the vector (x_axis_i, x_axis_j, x_axis_k). Those two vectors must be unit vectors within a tolerance of not more than 0.00001; an implementation may require a smaller tolerance.

It will be useful to have more compact notation, as follows (where the last two items are vectors):

x_axis_i = Xx
x_axis_j = Xy
x_axis_k = Xz
z_axis_i = Zx
z_axis_j = Zy
z_axis_k = Zz
origin_x = Tx
origin_y = Ty
origin_z = Tz
(Xx, Xy, Xz) = Vx
(Zx, Zy, Zz) = Vz

Define Vy as (Vz x Vx), meaning Vy is the cross product of Vz and Vx.

Define the components of Vy by Vy = (Yx, Yy, Yz). Then, by the definition of cross product:

Yx = (Zy x Xz) - (Zz x Xy)
Yy = (Zz x Xx) - (Zx x Xz)
Yz = (Zx x Xy) - (Zy x Xx)

The reason this works is that the Y-axis of a coordinate system is the cross product of the Z-axis with the X-axis. If the system is moved, that relationship continues to hold. Vy is the direction in which the transformed Y-axis points.

Now it is easy to build a 4x4 homogeneous coordinate transform matrix. The matrix is:

$$\begin{bmatrix} Xx & Yx & Zx & Tx \\ Xy & Yy & Zy & Ty \\ Xz & Yz & Zz & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To convert a point (x, y, z, 1) to its location in the transformed coordinate system, multiply as follows:

$$\begin{bmatrix} Xx & Yx & Zx & Tx \\ Xy & Yy & Zy & Ty \\ Xz & Yz & Zz & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

After a call to SET_COORDINATE_SYSTEM, all arguments to cmm_do_something functions (other than SET_COORDINATE_SYSTEM) which are interpreted in terms of a coordinate system, refer to the coordinate system described in the function call, until the end of a program or another call to SET_COORDINATE_SYSTEM.

The selected plane becomes the plane in the new coordinate system with the same name as the selected plane in the previously active coordinate system.

`void SET_DISTANCE_APPROACH(double distance)`

This sets the approach distance used in the MEASURE_POINT function. This function and the following four SET_DISTANCE_XXX functions implement the DMIS SNSET command [CAM-I, p. 150].

`void SET_DISTANCE_CLRSRF(double distance)`

This sets the clearance from surface distance used in probing. This distance is not currently used by the probing functions.

`void SET_DISTANCE_DEPTH(double distance)`

This sets the depth distance used in probing. This distance is not currently used by the probing functions.

`void SET_DISTANCE_RETRACT(double distance)`

This sets the retract distance used in the MEASURE_POINT function.

`void SET_DISTANCE_SEARCH(double distance)`

This sets the search distance used in the MEASURE_POINT function.

`void SET_FEED_RATE(double rate)`

All four canonical functions for setting feed rates (the others are SET_ROTARY_RATE, SET_SCAN_RATE, SET_TRAVERSE_RATE) take a numerical argument and assume a specific unit. All four are used in implementing the DMIS FEDRAT command [CAM-I, p. 173].

The rate is expressed in millimeters per minute. One alternative would be to have a "SET_VELOCITY_UNITS" command (the DMIS FEDRAT statement provides for setting units).

The feed rate is the constant rate of motion along whatever path is being followed relative to the object being probed. The CMM is expected to do its best to maintain this rate during measuring operations. The rate applies to the controlled point, which is usually the probe tip (maybe it should be the center of a sphere which is assumed to be at the end of the probe).

`void SET_PLANE(CANON_PLANE plane)`

Use the plane designated by "plane" as the selected plane. The selected plane must be the XY-plane, the XZ-plane, or the YZ-plane. This relates to the DMIS WKPLAN statement [CAM-I, p. 122]. The plane refers to one of the three principal planes of the current coordinate system.

`void SET_ROTARY_RATE(double rate)`

The rate is expressed in RPM of a rotary table relative to the housing or mounting of the rotary table. Also see notes for SET_FEED_RATE.

`void SET_ROTARY_ZERO(double angle)`

The angle of the rotary table is measured from some reference direction, (which has a default position) at which the angle is zero. This command moves the reference direction to the given angle, which is in terms of the default reference direction and is measured in current angle units (degrees in decimal form or radians). This implements the DMIS ROTSET command [CAM-I, p. 187].

After a call to this function, all arguments to cmm_do_something functions (other than SET_ROTARY_ZERO) which are rotary table angles are measured from the new reference direction, until the end of a program or another call to SET_ROTARY_ZERO.

Any value of "angle" is OK, provided it is a double.

`void SET_SCAN_DIST_INTERVAL`
`    (double dist_interval, CANON_AXIS axis)`

This functions and the other SET_SCAN_XXX functions, which follow, implement the DMIS SCNSET command [CAM-I, p. 245].

The dist_interval is expressed in current distance units.

The interval and axis are set as given and stay set until the end of the program or until another call to SET_SCAN_DIST_INTERVAL.

If axis is CANON_AXIS_NONE, the distance interval is measured "along the surface of the part from one point to the next." Otherwise the interval is measured "along the specified coordinate axis."

`void SET_SCAN_INTERVAL_TYPE(CANON_INTERVAL_TYPE interval_type)`

The interval_type is set as given and stays set until the end of the program or until another call to SET_SCAN_INTERVAL_TYPE. The interval_type and its corresponding interval are used for all scanning moves. Data is taken during a scanning move at the end of each interval. The degree of exactness of the interval is implementation-dependent.

`void SET_SCAN_RATE(double rate)`

The rate is expressed in millimeters per minute.

Also see notes for SET_FEED_RATE.

`void SET_SCAN_TIME_INTERVAL(double time_interval)`

The time_interval is expressed in seconds.

The interval is set as given and stays set until the end of the program or until another call to SET_SCAN_TIME_INTERVAL.

`void SET_SCAN_TYPE(CANON_SCAN_TYPE the_type)`

The scan type is set to the given type, which may be CANON_DRAG, CANON_NONCON, or CANON_PECK.

When the type is set to CANON_DRAG, scan moves are executed by dragging the sensor along the surface, in contact with the surface.

When the type is set to CANON_NONCON, scan moves are executed by any non-contact method. Examples of such methods are:

1. moving a capacitance probe along near the surface, but not in contact with the surface.

2. guiding a beam of electromagnetic radiation (light, radio waves, etc.) along the surface, and measuring by detecting reflected radiation.

When the type is set to CANON_PECK, scan moves are executed by moving the sensor along a line from the initial pose to the goal pose and intermittently moving the probe to touch the surface (with a touch trigger probe, for example) or to approach the surface but not touch it (with a capacitance probe, for example).

```
void SET_TRAVERSE_RATE(double rate)
```

The rate is expressed in millimeters per minute.

Also see notes for SET_FEED_RATE.

The traverse rate is the constant rate at which the CMM tries to move the controlled point during goto moves.

```
void STRAIGHT_TRAVERSE (double x, double y, double z)
```

Move at traverse rate in a straight line from the current point to the final XYZ position given by x, y, and z. Do not change the orientation of the probe during the move. This implements the DMIS GOTO command [CAM-I, p. 203]

```
void USE_ANGLE_UNITS(CANON_UNIT_ANGLE u)
```

Use the specified units for angle. The units must be one of the CANON_UNIT_ANGLE (see above). This function and the other two USE_XXX_UNITS functions, which follow, implement the DMIS UNITS command [CAM-I, p. 180].

```
void USE_LENGTH_UNITS(CANON_UNIT_LENGTH u)
```

Use the specified units for length. The units must be one of the CANON_UNIT_LENGTH (see above).

```
void USE_TEMPERATURE_UNITS(CANON_UNIT_TEMPERATURE u)
```

Use the specified units for temperature. The units must be one of the CANON_UNIT_TEMPERATURE (see above). The interpreter currently makes no use of temperature, so there is not much point in setting units.

B.5 Functions to Get Feature Parameters from Arrays of Points.

These functions are for the interpreter to call to get feature parameters from arrays of points. The points for a feature are assumed to be expressed in the same coordinate system as the one in which the nominal feature was defined. This is not necessarily the same as the currently active coordinate system.

There is one function for each implemented feature type defined in DMIS (circle, cylinder, line, plane, point).

Each function has an integer return value, which is to be used to pass status back to the calling function (either OK or ERROR). OK should be returned if the function was able to do its work without error. ERROR should be returned if the function detects problems, such as not enough points being provided to fully determine the feature.

Each function has two sets of parameters: input parameters and input/output parameters. All of these appear as arguments to the function.

Each function has three input parameters:

a. a pointer to an array of triples of (C++) doubles. Each triple represents a point (x, y, z).

b. an integer giving the number of points in the array to use.

c. a double giving the input_tolerance.

Each function has several input/output parameters, all of which are pointers to doubles and represent the parameters of a feature of the given type. Each of these must be set to its nominal value before the function is called. Each feature extraction function has the choice of using the nominal data or not using it. Likewise, each function may use the input_tolerance or not use it. Where several parameters are the components of a "normal" vector or a "direction" vector, these vectors must be unit vectors (within an implementation-dependent tolerance).

If the function returns OK, the values of these output doubles should have been set by the function. If the function returns error, the values of these doubles are meaningless after the function returns.

These functions are not required to make an effort to satisfy any constraints. It might be useful to add parameters to identify constraints. The most obvious constraint is to require that the sum of the squares of the errors be minimized. Another reasonable constraint would be to require the function to try to minimize the maximum error. A third possibility is to use the nominal values to constrain the assignment of output parameters. For example, if all points lie within the input_tolerance of the nominal feature, the function might be constrained to return the nominal feature.

It might be useful to allow several alternative functions where only one is prototyped here. This could be done by adding a parameter which is the name (or other identifier) of a function which can do the required work.

It would also be feasible to have a single function to do the work of all the functions here. It would have additional input arguments to indicate the feature type and other things. The meaning of the output arguments would vary according to the input feature type. This does not seem like a good thing to do.

There are other interesting issues.

1. The DMIS ALGDEF statement allows the naming of algorithms in an array of algorithms, and the GEOALG statement allows assigning an algorithm (identified by type names) for use in generating parameters for features of a given type from point sets. These have not been implemented but might be.

2. DMIS allows for the simultaneous determination of all parameters of an "actual" feature, but does not allow for the parameters to be given actual values one at a time. This is very strange. The DMIS OBTAIN statement allows a variable to be given the value of a parameter of a feature. What is needed is the reverse operation (SETPAR?) of giving a single parameter of a feature an actual value. If SETPAR were available, measurement algorithms could be readily written in

DMIS. As DMIS is, it is hard to write measurement algorithms in DMIS.

```
int extract_circle(double points [][3], int how_many,
    double tolerance, double * center_x, double * center_y,
    double * center_z, double * normal_i, double * normal_j,
    double * normal_k, double * diameter)
```

extract_circle finds a circle fit to the points. It produces the coordinates of the center of the circle, the direction of the normal to the plane of the circle, and the diameter of the circle.

```
int extract_cylinder(double points [][3], int how_many,
    double tolerance, double * point_x, double * point_y,
    double * point_z, double * direction_x, double * direction_y,
    double * direction_z, double * diameter)
```

extract_cylinder finds a cylinder fit to the points. It produces the coordinates of a point on the axis of the cylinder, the direction of the axis, and the diameter of the cylinder.

```
int extract_line(double points [][3], int how_many,
    double tolerance, double * point_x, double * point_y,
    double * point_z, double * direction_x,
    double * direction_y, double * direction_z)
```

extract_line finds a line fit to the points. It produces the coordinates of a point on the line and the direction of the line.

```
int extract_plane(double points [][3], int how_many,
    double tolerance, double * point_x, double * point_y,
    double * point_z, double * normal_i, double * normal_j,
    double * normal_k)
```

extract_plane finds a plane fit to the points. It produces the coordinates of a point on the plane and a normal to the plane.

```
int extract_point(double points [][3], int how_many,
    double tolerance, double * point_x, double * point_y,
    double * point_z)
```

extract_point finds a single point fit to the data points.

Appendix C Building a Stand-Alone Executable

On a SUN SPARCstation 20, an executable file for the stand-alone interpreter may be built from source code in under two minutes, as described below. The same procedure should work on any computer running a Unix operating system that has the standard C++ libraries. On computers running other operating systems, including PCs, compilation should also be easy, provided the standard C++ libraries are available.

To make an executable, nine source code files must be placed in the same directory along with the Makefile shown in Table 5 below. The source code files are:

cmm_canon.cc - function definitions for four of five interfaces to interpreter
cmm_canon.hh - header file for all interfaces to interpreter
cmm_canon_extract1.cc - function definitions for extract_feature interface to interpreter
driver.cc - user interface
fit.hh - header file for feature extraction functions
interp.cc - function definitions for interpreter kernel
interp.hh - header file for interpreter kernel
rest_world.hh - header file for dummy world model
tk_common.hh - header file with constants and aliases for clearer programming

An executable file named “dmis” is built in the same directory by giving the command:
make dmis

To use the Makefile, the definitions of COMPILE and LINK, must be changed to be correct in the environment where the compilation is being done.

In the Makefile, we are using the Centerline C++ compiler. The Gnu C++ compiler has also been used.

```
COMPILE = CC -c -v -g -O
LINK = CC -v
DMIS_O = driver.o interp.o cmm_canon.o cmm_canon_extract.o
cmm_canon.o: cmm_canon.cc cmm_canon.hh interp.hh
	$(COMPILE) cmm_canon.cc
cmm_canon_extract.o: cmm_canon_extract1.cc fit.hh interp.hh cmm_canon.hh
	$(COMPILE) +a1 -o cmm_canon_extract.o cmm_canon_extract1.cc
dmis: $(DMIS_O)
	$(LINK) -o dmis $(DMIS_O) -lm
driver.o: driver.cc rest_world.hh cmm_canon.hh tk_common.hh
	$(COMPILE) driver.cc
interp.o: interp.cc interp.hh cmm_canon.hh tk_common.hh
	$(COMPILE) interp.cc
```

Table 5. Makefile for Interpreter

Appendix D Transcript of a Session

This is a transcript of a session using the stand-alone interpreter. Characters entered by the user are shown in **boldface**. All user input is followed by a carriage return not shown here.

```
1} dmis
enter a command or “help”
CMD => help
COMMANDS
--------------------------------------------
help
quit
interp_init()
interp_open_program(“program file name”)
interp_execute_next()
interp_close_program()
interp_exit()
run_program(“program file name”)
CMD => interp_open_program(“programs/short4.dms”)
interpreter not initialized, cannot open program
CMD => interp_init()
initializing dmis interpreter
CMD => interp_open_program(“programs/short4.dms”)
CMD => interp_execute_next()
    1     N1 PROGRAM_START(“hi mom”)
CMD => interp_execute_next()
    2     N2 USE_LENGTH_UNITS(CANON_UNIT_INCH)
    3     N2 USE_ANGLE_UNITS(CANON_UNIT_ANGDEC)
CMD => interp_execute_next()
    4     N3 ADVISORY(“Interpreted MODE”)
CMD => interp_execute_next()
    5     N4 PROGRAM_END()
CMD => interp_execute_next()
last statement has been executed - close program
CMD => interp_close_program()
closing current program
CMD => run_program("programs/short4.dms")
    1     N1 PROGRAM_START("hi mom")
    2     N2 USE_LENGTH_UNITS(CANON_UNIT_INCH)
    3     N2 USE_ANGLE_UNITS(CANON_UNIT_ANGDEC)
    4     N3 ADVISORY("Interpreted MODE")
    5     N4 PROGRAM_END()
CMD => interp_exit()
exiting dmis interpreter
```

Appendix E Error Handling and Error Messages

E.1 Error Handling

The interpreter detects and flags most kinds of illegal input. For example, unreadable input, missing words, extra words, out-of-bounds numbers, and illegal combinations of words are all detected. The interpreter does not check for axis overtravel or excessively high feeds or speeds, or situations where a legal command does something unfortunate, such as crashing a probe.

The basic approach to error handling is:

1. Check carefully for errors.
2. If an error occurs, identify it specifically so that the user can be informed.
3. If an error occurs, return through the function call hierarchy rather than jumping out of it.

Errors are handled somewhat differently during DMIS file reading and execution. In either case, a function detecting an error or receiving an ERROR returned value does no further processing, but returns ERROR (or ERRD or ERRN). The returned value of ERROR is passed up the calling chain, reaching the top of the chain at interp_open_program or interp_execute_next. Error handling during file reading differs from error handling during execution only in the timing and method of printing error messages.

Interp_open_program is the interface function that triggers DMIS file reading. During a call to that function, if an error is detected, one message is printed immediately. Then ERROR is returned up the chain of function calls until the build_program function is reached. It prints the line of code being read when the error occurred; then it returns ERROR to interp_open_program, which returns ERROR to the caller.

Interp_execute_next is the interface function that causes the next DMIS statement to be executed. During a call to interp_execute_next, if an error is detected, the message is recorded. Then ERROR is passed up the function call chain until interp_execute_next is reached. At that point the error message is printed and ERROR is returned to the caller.

Most error messages that can be sent during a call to interp_open_program are listed in section E.3 below. In addition to those listed, messages may be generated by the Lex scanner or the YACC parser. A few types of DMIS program syntax error will result in Lex scanner error messages, which are short and do not explain much. Most DMIS program syntax errors will cause an error in the YACC parser. When this occurs, the parser will print a line describing what it was looking for when the error occurred. For example, if “ANGDEC” is misspelled as “ANGDEW” in a UNITS statement, the following line is printed.

```
parse error, expecting `ANGDEC' or `ANGDMS' or `ANGRAD'
```

E.2 Sources of Error Messages

Most error messages sent by interpreter come from kernel functions when bad input or bad sensory data is detected. These are listed in section E.3.

In addition:

1. The interpreter driver, which is used in the stand-alone interpreter only, also has a few input error messages. They appear if the user gives an inappropriate command and are not

covered here.

2. The extraction routine for each feature type has one error message, which simply says the routine failed.
3. The interp_do_something functions include one error message, "probing failed," which will be printed if probing fails.

E.3 Error Messages

Error messages generated by the kernel during execution are listed below. Most of the messages represent additional checks on the input which the parser will not detect. Some of the messages should never be triggered by any input because the errors for which they check will be detected sooner by the parser. Most of the checks that result in never triggered error messages are made by the "else" at the end of an "if, else if, ..., else if, else" construct in the C++ source code, where the "if" and "else if's" are intended to be exhaustive of all possibilities.

This is a list of all 162 error messages in the interpreter kernel. The list is arranged alphabetically. Messages are in **boldface** type. Following each message is the name of the function or functions in which it is found, printed in *italics*. Italic text inside an error message indicates that some variable value will appear in that place when the message is printed.

1. *axis type* **is not valid with a cartesian point** . . . . *output_actual_tol_cortol*
2. *axis type* **is not valid with a polar point** . . . . *output_actual_tol_cortol*
3. ENDGO position differs from GOTARG position . . . . *convert_endgo*
4. actual circle does not exist . . . . *output_actual_circle*
5. actual cylndr does not exist . . . . *output_actual_cylndr*
6. actual line does not exist . . . . *find_rotation_angle, output_actual_line*
7. actual plane does not exist . . . . *output_actual_plane*
8. actual point does not exist . . . . *output_actual_point*
9. actual x-origin plane does not exist . . . . *build_transform_dat*
10. actual y-origin plane does not exist . . . . *build_transform_dat*
11. actual z-origin plane does not exist . . . . *build_transform_dat*
12. angle *number* **more than a full circle** . . . . *convert_rotset*
13. angle between nominal and actual lines > *number* **degrees** . . . . *extract_line_cart_bnd,*
. . . . *extract_line_cart_unbnd*
14. angle between nominal and actual normal > *number* **degrees** . . . . *extract_plane_cart*
15. array is full, cannot continue . . . . *record_item*
16. attempt to turn off MAN mode . . . . *convert_mode*
17. axis to align more than *number* **degrees out of plane** . . . . *find_rotation_aux*
18. bad SNSET command . . . . *convert_snset*
19. bad axis type . . . . *complete_transform, convert_scnset, find_rotation_aux,*
. . . . *output_rotate, rotate_matrix*
20. bad block head for endgo . . . . *convert_endgo*
21. bad block head for endmes . . . . *convert_endmes*
22. bad data type for a variable . . . *make_variable, read_dmis_variables, write_dmis_variables*
23. bad data type name *name* . . . . *read_dmis_variables*
24. bad direction_x . . . . *convert_ptmeas*
25. bad direction_y . . . . *convert_ptmeas*
26. bad direction_z . . . . *convert_ptmeas*

70. feature not a point . . . . . *convert_meas_point*
71. file name is null . . . . . *make_filnam*
72. first item in block's item list is not a line . . . . . *find_next_line*
73. first line of program is not dmismn . . . . . *find_first_line*
74. item doubly defined . . . . . *record_item*
75. item is neither a line nor a block . . . . . *find_next_line*
76. item stack is empty . . . . . *find_next_line*
77. item stack not properly initialized . . . . . *find_first_line*
78. last branch point is not a meas_block . . . . . *make_endmes*
79. last meas_block does not match endmes . . . . . *make_endmes*
80. line has no name . . . . . *make_feat_line*
81. line is bound . . . . . *output_actual_tol_cortol*
82. line parallel to plane, or nearly so . . . . . *intersect_plane_line*
83. line twin does not exist . . . . . *find_feature_direction, output_actual_tol_cortol*
84. negative angle *number* **for rotary table** . . . . . *convert_rotab, convert_rotset*
85. negative velocity given in fedrat . . . . . *convert_fedrat*
86. normal vector length not 1 . . . . . *extract_circle_cart, extract_plane_cart*
87. null or empty string variable name . . . . . *make_variable*
88. null text string . . . . . *make_text*
89. null variable name . . . . . *make_decl, read_dmis_variables*
90. number of features and actualities differ . . . . . *convert_const*
91. opposed directions used in datset . . . . . *make_datset_dat*
92. parameter number *number* **out of bounds** . . . . . *find_feature_parameter_value*
93. plane has no name . . . . . *make_feat_plane*
94. plane twin does not exist . . . . . *find_feature_direction, find_feature_parameter_value,*
. . . . . *find_feature_point, output_actual_tol_cortol*
95. planes parallel or nearly so . . . . . *intersect_two_planes*
96. point has no name . . . . . *make_feat_point*
97. point twin does not exist . . . . . *find_feature_parameter_value, find_feature_point,*
. . . . . *output_actual_tol_cortol*
98. probe_cart name is null . . . . . *make_probe_cart*
99. probe_pol name is null . . . . . *make_probe_pol*
100. program missing from interp model . . . . . *close_program*
101. program variable *name* **has no saved value** . . . . . *read_dmis_variables*
102. program variable *name* **type isn't stored type** . . . . . *read_dmis_variables*
103. reference to *name* **not found** . . . . . *find_feature_all, find_reference, find_variable*
104. rotary table angle too large . . . . . *convert_rotab*
105. rotate_axes_abs has no name . . . . . *make_rotate_axes_abs*
106. rotate_axes_rel has no name . . . . . *make_rotate_axes_rel*
107. rotdef has no name . . . . . *make_rotdef*
108. same axis used twice in datset . . . . . *make_datset_dat*
109. same datum used for two directions in datset . . . . . *make_datset_dat*
110. scnset type is not DIST or TIME . . . . . *convert_scnset*
111. string not allocated for dmismn . . . . . *make_dmismn*
112. thldef name is null . . . . . *make_thldef*
113. tol name is null . . . . . *insert_tol_name*

114. too few points (*number*) for circle. . . . . *convert_const_circle, convert_meas_circle*
115. too few points (*number*)for cylndr . . . . *convert_const_cylndr, convert_meas_cylndr*
116. too few points (*number*) for line . . . . *convert_const_line, convert_meas_line*
117. too few points (*number*) for plane . . . . *convert_const_plane, convert_meas_plane*
118. trans has no name . . . . *make_trans*
119. transform missing . . . . *convert_recall*
120. two UNITS lines used . . . . *make_units*
121. two rotary tables defined . . . . *make_rotdef*
122. unable to open file *name* for reading . . . . *, preprocess_dmis, read_dmis_variables,*
. . . . *read_in_dmis*
123. unable to open file *name* for writing . . . . *convert_filnam, preprocess_dmis,*
. . . . *write_dmis_variables*
124. unknown angle_unit_type . . . . *convert_units*
125. unknown command. . . . . *execute_next_line*
126. unknown const type . . . . *convert_const*
127. unknown coordinate type. . . . . *output_actual_tol_cortol*
128. unknown datset subtype. . . . . *convert_datset*
129. unknown definition command . . . . *.convert_definition*
130. unknown feature type. . . . . *convert_feat, output_actual_feature,*
. . . . *output_actual_tol_cortol, output_actual_tol_diam*
131. unknown feed_set_type . . . . *convert_fedrat*
132. unknown goto type . . . . *.convert_goto*
133. unknown length_unit_type. . . . . *convert_units*
134. unknown line_other command . . . . *convert_line_other*
135. unknown meas subtype. . . . . *convert_endmes*
136. unknown on_off type . . . . *convert_prcomp*
137. unknown origin type in datset . . . . *save_datset_orig*
138. unknown point type . . . . *output_actual_tol_cortol*
139. unknown setting command . . . . *.convert_setting*
140. unknown temperature_unit_type . . . . *convert_units*
141. unknown tolerance type . . . . *output_actual_tolerance*
142. unknown vform type. . . . . *save_disply_item*
143. unknown wkplan_type . . . . *convert_wkplan*
144. unset value obtained . . . . *convert_obtain*
145. unusable feature type . . . . *.find_feature_direction, find_feature_parameter_value,*
. . . . *find_feature_point*
146. using unset variable double . . . . *.code_header*
147. variable doubly defined . . . . *record_variable*
148. variable is not type double . . . . *make_assign, make_obtain*
149. variable not of type *type* . . . . *make_ex_double_variable*
150. variables array is full, cannot continue. . . . . *record_variable*
151. velocity needed but missing in fedrat . . . . *convert_fedrat*
152. velocity provided in fedrat - should not be . . . . *convert_fedrat*
153. wrong number (*number*) of points measured . . . . *convert_endmes*
154. wrong number of point measurements . . . . *make_endmes*
155. wrong number of points (*number*) for point - must be 1 . . . . *convert_meas_point*

156. x origin used twice in DATSET . . . . *save_datset_orig*
157. x origin used twice in TRANS . . . .*dmis_yacc*
158. y origin used twice in DATSET . . . . *save_datset_orig*
159. y origin used twice in TRANS . . . .*dmis_yacc*
160. z origin used twice in DATSET . . . . *save_datset_orig*
161. z origin used twice in TRANS . . . .*dmis_yacc*
162. zero length direction vector used. . . . .*convert_goto*

Appendix F YACC and Lex Specifications

F.1 Introduction

The YACC and Lex specifications used by the DMIS interpreter are discussed in this section.The YACC specification does not deal with comments or line continuations. The interpreter runs the DMIS programs through a pre-processor to remove comments and join lines which are continued. The parser built from this YACC specification takes the pre-processed file as input.

F.2 Lex Scanner

The lex scanner that works with the YACC specification is straightforward. White space (tab or space) is allowed between all groups of characters that are tokens. Letters in groups of characters that make tokens are treated the same in upper case or lower case, as stipulated by the DMIS spec. All the tokens are spelled as shown in the section below that lists tokens, except as follows. In the items below, an alphanumeric character is a letter, digit, or underscore.

F.2.1 Changes from the First Version

1. The definition of a label was revised since integers may be used as labels.

2. ASSIGN, BOOL, COMMON, DECL, DOUBLE, INTGR, and OBTAIN were added.

3. The ability to deal with parentheses around expressions was added.

F.2.2 Summary of Lex Rules

1. All tokens that are DMIS major words and take a following slash are spelled with the slash included.

2. COMMA_V is a comma followed by a V.

3. D2 is spelled “2D”.

4. D3 is spelled “3D”.

5. LABEL_OR_INTEGER is one or more digits.

6. LABEL_OR_VARIABLE is a letter followed by zero to five alphanumeric characters.

7. LABEL_SURE is an alphanumeric character followed by one to nine characters, each of which is an alphanumeric character or a minus sign (dash). As defined, some LABEL_SUREs may also be a LABEL_OR_INTEGER. LABEL_SURE will not be recognized when there is a choice because it is defined later in the lex file.

8. REAL is an optional plus or minus sign followed by zero to many digits, followed by an optional decimal point, followed by zero to many digits (provided that there is at least one digit somewhere in the number). As defined, some sequences of digits qualify as a REAL, a LABEL_SURE, and a LABEL_OR_INTEGER. When a sequence of digits is read that could be a REAL or something else, REAL will not be recognized because it is defined last in the lex file.

10. TEXT_STRING is a single quote, followed by any number of characters that are not single quotes, followed by a single quote.

11. X_DIR is spelled “-XDIR”.

12. Y_DIR is spelled “-YDIR”.

13. Z_DIR is spelled “-ZDIR”.

F.3 YACC

These YACC grammar rules do not include all constraints included in DMIS. Some DMIS statements that are readable under these grammar rules will not be executable because they violate constraints. Almost all constraint violations will be detected by the interpreter and will result in error messages. The error messages are included in Appendix E.

The topmost grammatical unit is “program,” so the entire DMIS program is read at once. The first grammar rule is for “program”. The rest of the rules are arranged alphabetically.

The action portions of the YACC rules have been deleted since they are C++ code which is difficult to understand. The full YACC specification with actions included is available on request.

F.3.1 Changes from First Version

The differences between this YACC specification and the one used for the first version of the interpreter and shown in [Kramer2] were driven primarily by the need to implement variables in Version 2. In addition, some tidying up was done.

1. The definition of a program has been revised to by (1) requiring that if FILNAM is used, it must be the second statement and (2) if DECL is used for declaring variables, it must be the third statement if FILNAM is used or the second statement, if not.

2. The return value of an action may be of various data types. In the first version, all returned values were doubles.

3. Almost everywhere “real” was used in the first version, Version 2 uses ex_real (meaning a real-valued expression).

4. The DMIS main words ASSIGN, DECL, and OBTAIN were added to the specification.

5. The DMIS secondary words BOOL, COMMON, DOUBLE, and INTGR were added to the specification.

6. All C++ code not part of an action has been removed and placed in interp.cc.

7. Almost all actions have been changed to use function calls where the first version may have used several lines of low-level C++ code.

F.3.2 Formal Specification

```
%union {
  int bval; /* compiler complains if BOOL is used in place of int */
  double dval;
  int ival;
  ex_double * exdptr;
}


%type <bval> d_type
        s_type

%type <dval> real

%type <exdptr> ex_real

%type <ival> angle_unit axis_id axis_type
        bound_type
        data_type  dimension_type  disply_type
        f_type  feature_type
        in_out_type
        length_unit
        on_off_type  origin  output_destination
        plane  point_type  probe_type
        rotary_axis
        scnset_type  snset_type
        temperature_unit  tol_code
        update_type
        velocity_type   velocity_unit
        wise_type_absl  wise_type_incr


/* the following types are needed to make bison happy, but are not used */

%type <ival> datset_items
        f_item  f_list


%start program

%token <ival> A ABSL ACT ALL AMT ANGDEC ANGDMS ANGLE ANGRAD
%token <ival>   APPRCH ARC ASSIGN AUTO AVG
%token <ival> BF BND BOOL
%token <ival> CART CCW CHORD CIRCLE CLRSRF CM
%token <ival>   COMM COMMA_V COMMON CONST CORTOL CW CYLCTY CYLNDR
%token <ival> D DA DAT DATDEF DATSET DECL DEFALT DEPTH DEV
%token <ival>   DIAM DISPLY DIST DMIS DMISMN DOUBLE DRAG D2 D3
%token <ival> ENDFIL ENDGO ENDMES
%token <ival> F FA FEAT FEDRAT FEET FILNAM FIXED FLAT FORCE
```

```
%token <ival> GOTARG GOTO
%token <ival> HIGH HIST
%token <ival> INCH INCR INDEX INNER INTGR IPM
%token <ival> LABEL_OR_INTEGER LABEL_OR_VARIABLE LABEL_SURE LINE LMC
              LOW
%token <ival> M MAN MCS MEAS MESVEL MM MMC MODE MPM
%token <ival> NOM NONCON
%token <ival> OBTAIN OFF ON OPER OUTER OUTFIL OUTPUT
%token <ival> PARLEL PCENT PECK PERP PLANE PLOT POINT POL POS
%token <ival>   POSVEL PRCOMP PRINT PROBE PROG PTMEAS
%token <ival> RADIUS
%token <dval>  REAL
%token <ival>  RECALL RETRCT RFS ROTAB ROTATE ROTDEF
%token <ival>   ROTNUL ROTORG ROTSET ROTTOT ROTVEL RPM RT
%token <ival> S SA SAVE SCNMOD SCNSET SCNVEL SEARCH SHORT
%token <ival>   SNSDEF SNSET SNSLCT STAT STOR
%token <ival> T TA TEMPC TEMPF TERM TEXT TEXT_STRING
%token <ival>   TH THLDEF TIME TOL TRANS
%token <ival> UNBND UNITS
%token <ival> V VFORM
%token <ival> WKPLAN
%token <ival> X_DIR XAXIS XDIR XORIG XYPLAN
%token <ival> Y_DIR YAXIS YDIR YORIG YZPLAN
%token <ival> Z_DIR ZAXIS ZDIR ZORIG ZXPLAN
%%


program : dmismn filnams decls blocks endfil;

angle_unit : ANGDEC
        | ANGDMS
        | ANGRAD;

assign : LABEL_OR_VARIABLE '=' ASSIGN ex_real '\n';

axis_id : XDIR
        | X_DIR
        | YDIR
        | Y_DIR
        | ZDIR
        | Z_DIR;

axis_type : ANGLE
        | RADIUS
        | XAXIS
        | YAXIS
        | ZAXIS;
```

```
block       : one_liner
            | many_liner;

blocks      : /* empty */
            | blocks block;

bound_type : BND
            | UNBND;

const_bf : CONST feature_type ‘,’ F label ‘,’ BF ‘,’ FA label ‘,’ f_list ‘\n’;

d_type      : D
            | DA;

data_type : BOOL
            | DOUBLE
            | INTGR;

datdef      : DATDEF f_type label ‘,’ DAT label ‘\n’;

datset      : D label ‘=’ DATSET MCS ‘\n’
            | D label ‘=’ DATSET datset_items;

datset_do : datset_dir
            | datset_do ‘,’ datset_orig;

datset_dir : XDIR
            | X_DIR
            | YDIR
            | Y_DIR
            | ZDIR
            | Z_DIR;

datset_items : DAT label ‘,’ datset_do ‘\n’
            | DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_do ‘\n’
            | DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_origs ‘\n’
            | DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_origs ‘\n’
            | DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_origs ‘,’ DAT label ‘,’ datset_origs ‘\n’
            | DAT label ‘,’ datset_do ‘,’ DAT label ‘,’ datset_origs ‘,’ DAT label ‘,’ datset_do ‘\n’;

datset_origs : datset_orig
            | datset_origs ‘,’ datset_orig;

datset_orig : XORIG
            | YORIG
            | ZORIG;

decl        : DECL COMMON ’,’ data_type ’,’ variable_names ’\n’;

decls       : /* empty */
            | decls decl;
```

```
definition : datdef
        | feat
        | thldef
        | probe_definition
        | rotdef
        | tolerance_definition;

dimension_type : D2
        | D3;

disply  : DISPLY OFF ‘\n’
        | DISPLY disply_list ‘\n’;

disply_item : disply_type ‘,’ DMIS
        | disply_type COMMA_V label
        | disply_type ‘,’ DMIS COMMA_V label;

disply_list : disply_item
        | disply_list ‘,’ disply_item;

disply_type : COMM
        | PRINT
        | STOR
        | TERM;

dmismn  : DMISMN TEXT_STRING ‘\n’;

endfil  : ENDFIL;

endgo   : ENDGO;

endmes  : ENDMES;

ex_real : ’(’ LABEL_OR_VARIABLE ’)’
        | real
        | ’(’ ex_real ’)’;

f_item  : f_type label;

f_list  : f_item
        | f_list ‘,’ f_item;

f_type  : F
        | FA;

feat_circle : F label ‘=’ FEAT CIRCLE ‘,’ in_out_type ‘,’ point_type ‘,’
            ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘\n’;

feat_cylndr : F label ‘=’ FEAT CYLNDR ‘,’ in_out_type ‘,’ point_type ‘,’
            ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘\n’
        | F label ‘=’ FEAT CYLNDR ‘,’ in_out_type ‘,’ point_type ‘,’ ex_real ‘,’
            ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘,’ ex_real ‘\n’;
```

```
feat_line : F label '=' FEAT LINE ',' bound_type ',' point_type ','
            ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real ','
            ex_real ',' ex_real ',' ex_real '\n';

feat_plane : F label '=' FEAT PLANE ',' point_type ',' ex_real ',' ex_real ',' ex_real ','
            ex_real ',' ex_real ',' ex_real '\n';

feat_point : F label '=' FEAT POINT ',' point_type ','
            ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real '\n';

feat       : feat_circle
           | feat_cylndr
           | feat_line
           | feat_plane
           | feat_point;

feature_type : CIRCLE
           | CYLNDR
           | LINE
           | PLANE
           | POINT;

fedrat     : FEDRAT velocity_type ',' velocity_unit '\n'
           | FEDRAT velocity_type ',' velocity_unit ',' ex_real '\n';

filnam     : FILNAM TEXT_STRING '\n';

gotarg     : GOTARG ex_real ',' ex_real ',' ex_real '\n';

gotarg_block : gotarg gotos endgo;

goto       : GOTO ex_real ',' ex_real ',' ex_real '\n'
           | GOTO INCR ',' ex_real ',' ex_real ',' ex_real ',' ex_real '\n';

gotos      : goto       /* specs require at least one, p. 202 */
           | gotos goto;

in_out_type : INNER
           | OUTER;

label      : '(' LABEL_SURE ')'
           | '(' LABEL_OR_VARIABLE ')'
           | '(' LABEL_OR_INTEGER ')';

length_unit : CM
           | FEET
           | INCH
           | M
           | MM;

many_liner : meas_block
           | gotarg_block;

meas       : MEAS feature_type ',' F label ',' LABEL_OR_INTEGER '\n';
```

```
meas_block : meas meas_goes endmes;

meas_goes : /* empty */
          | meas_goes ptmeas
          | meas_goes goto;

mode      : MODE MAN '\n'
          | MODE PROG ',' MAN '\n'
          | MODE AUTO ',' MAN '\n'
          | MODE AUTO ',' PROG ',' MAN '\n';

obtain : LABEL_OR_VARIABLE '=' OBTAIN f_type label ',' LABEL_OR_INTEGER '\n';

on_off_type : ON
          | OFF;

one_liner : const_bf
          | definition
          | disply
          | goto
          | output
          | recall
          | rotab
          | rotate
          | save
          | setting
          | snslct
          | text
          | trans
          | units
          | vform;

origin    : XORIG
          | YORIG
          | ZORIG;

output    : OUTPUT FA label ta_list '\n';

output_destination : MAN
          | OPER
          | OUTFIL;

plane     : XYPLAN
          | YZPLAN
          | ZXPLAN;

point_type : CART
          | POL;

prcomp    : PRCOMP on_off_type '\n';
```

```
probe_definition : S label '=' SNSDEF PROBE ',' probe_type ','
                   point_type ',' ex_real ',' ex_real ',' ex_real ',' ex_real ','
                   ex_real ',' ex_real ',' ex_real '\n';

probe_type : FIXED
           | INDEX;

ptmeas     : PTMEAS point_type ',' ex_real ',' ex_real ',' ex_real '\n'
           | PTMEAS point_type ',' ex_real ',' ex_real ',' ex_real ',' ex_real ','
                   ex_real ',' ex_real '\n';

real       : LABEL_OR_INTEGER
           | REAL;

recall     : RECALL d_type label '\n'
           | RECALL FA label '\n'
           | RECALL s_type label '\n'
           | RECALL RT label '\n';

rotab      : ROTAB RT label ',' INCR ',' wise_type_incr ',' update_type ',' ex_real '\n'
           | ROTAB RT label ',' ABSL ',' wise_type_absl ',' update_type ',' ex_real '\n';

rotary_axis : XAXIS
           |  YAXIS
           |  ZAXIS;

rotdef : RT label '=' ROTDEF ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real ',' ex_real '\n';

rotset : ROTSET RT label ',' ex_real '\n';

rotate     : D label '=' ROTATE rotary_axis ',' ex_real '\n'
           | D label '=' ROTATE rotary_axis ',' f_type label ',' axis_id '\n';

s_type     : S
           | SA;

save       : SAVE d_type label '\n'
           | SAVE FA label '\n'
           | SAVE s_type label '\n'
           | SAVE RT label '\n';

scnmod     : SCNMOD  on_off_type '\n';

scnset     : scnset_dist
           | scnset_chord
           | scnset_time
           | scnset_angle
           | scnset_defalt;

scnset_angle : SCNSET scnset_type ',' ANGLE ',' ex_real '\n';

scnset_chord : SCNSET scnset_type ',' CHORD ',' ex_real '\n'
           | SCNSET scnset_type ',' CHORD ',' ex_real ',' ex_real '\n';

scnset_defalt : SCNSET scnset_type ',' DEFALT '\n';
```

```
scnset_dist : SCNSET scnset_type ‘,’ DIST ‘,’ ex_real ‘\n’
        | SCNSET scnset_type ‘,’ DIST ‘,’ ex_real ‘,’ rotary_axis ‘\n’;

scnset_time : SCNSET scnset_type ‘,’ TIME ‘,’ ex_real ‘\n’;

scnset_type : PECK
        | DRAG
        | NONCON;

setting : assign
        | datset
        | fedrat
        | mode
        | obtain
        | prcomp
        | rotset
        | scnmod
        | scnset
        | snset
        | wkplan;

snset   : SNSET snset_type ‘,’ ex_real ‘\n’;

snset_type : APPRCH
        | CLRSRF
        | DEPTH
        | RETRCT
        | SEARCH;

snslct : SNSLCT S label ‘\n’;

ta_item :  ‘,’ TA label;

ta_list : /* empty */
        | ta_list ta_item;

temperature_unit : TEMPC
        |  TEMPF;

text    : TEXT output_destination ‘,’ TEXT_STRING ‘\n’;

thldef : TH label ‘=’ THLDEF S label ‘,’ LABEL_OR_INTEGER ‘\n’;

tolerance_definition : tol_cortol
        | tol_cylcty
        | tol_diam
        | tol_flat
        | tol_parlel
        | tol_perp
        | tol_pos;
```

```
tol_code : LMC
        | MMC
        | RFS;

tol_cortol : T label ‘=’ TOL CORTOL ‘,’ axis_type ‘,’ ex_real ‘,’ ex_real ‘\n’;

tol_cylcty : T label ‘=’ TOL CYLCTY ‘,’ ex_real ‘\n’;

tol_diam : T label ‘=’ TOL DIAM ‘,’ ex_real ‘,’ ex_real ‘\n’
        | T label ‘=’ TOL DIAM ‘,’ ex_real ‘,’ ex_real ‘,’ AVG ‘\n’;

tol_flat : T label ‘=’ TOL FLAT ‘,’ ex_real ‘\n’;

tol_parlel : T label ‘=’ TOL PARLEL ‘,’ ex_real ‘,’ tol_code ‘,’ DAT label ‘,’ tol_code ‘\n’;

tol_perp : T label ‘=’ TOL PERP ‘,’ ex_real ‘,’ tol_code ‘,’ DAT label ‘,’ tol_code ‘\n’;

tol_pos : tol_pos0
        | tol_pos1
        | tol_pos2
        | tol_pos3;

tol_pos0 : T label ‘=’ TOL POS ‘,’ dimension_type ‘,’ ex_real ‘,’ tol_code ‘\n’;

tol_pos1 : T label ‘=’ TOL POS ‘,’ dimension_type ‘,’ ex_real ‘,’ tol_code ‘,’
             DAT label ‘,’ tol_code ‘\n’;

tol_pos2 : T label ‘=’ TOL POS ‘,’ dimension_type ‘,’ ex_real ‘,’ tol_code ‘,’
             DAT label ‘,’ tol_code ‘,’ DAT label ‘,’ tol_code ‘\n’;

tol_pos3 : T label ‘=’ TOL POS ‘,’ dimension_type ‘,’ ex_real ‘,’ tol_code ‘,’
             DAT label ‘,’ tol_code ‘,’ DAT label ‘,’ tol_code ‘,’ DAT label ‘,’ tol_code ‘\n’;

trans   : D label ‘=’ TRANS trans_spec ‘\n’
        | D label ‘=’ TRANS trans_spec ‘,’ trans_spec ‘\n’
        | D label ‘=’ TRANS trans_spec ‘,’ trans_spec ‘,’ trans_spec ‘\n’;

trans_spec : origin ‘,’ ex_real
        | origin ‘,’ f_type label;

units   : UNITS length_unit ‘,’ angle_unit ‘\n’
        | UNITS length_unit ‘,’ angle_unit ‘,’ temperature_unit ‘\n’;

update_type : ROTNUL
        | ROTORG
        | ROTTOT;

variable_name : LABEL_OR_VARIABLE;

variable_names : variable_name
        | variable_names ’,’ variable_name;

velocity_type : MESVEL
        | POSVEL
        | ROTVEL
        | SCNVEL;
```

```
velocity_unit : DEFALT
          | HIGH
          | IPM
          | LOW
          | MPM
          | PCENT
          | RPM;

vform     : V label '=' VFORM v_list '\n';

v_list    : v_type
          | v_list ',' v_type;

v_type    : ACT
          | ALL
          | AMT
          | DEV
          | HIST
          | NOM
          | PLOT
          | STAT;

wise_type_absl : CCW
          | CW
          | SHORT;

wise_type_incr : CCW
          | CW;

wkplan    : WKPLAN plane '\n';

%%
```